8 Developing Large Systems in Small Steps

Ulrik Pettersson

This chapter is an attempt to summarize experiences, useful practices, and lessons learnt from the development of large systems. These experiences have been gained mainly from the development of telecommunication systems at Ericsson, but also from the development of avionics systems at Saab Aeronautics.

Introduction

Systems such as a node in a telecommunication network and an avionics system in an aircraft differ when considering function and techniques. But the things they have in common are very typical for large systems, and may actually be the things that make large systems so tricky to develop. Large systems:

- Evolve over many years, sometimes even decades, with a new release usually once or twice a year.
- Are made up from thousands of software and hardware components that exist in many versions and variants.
- Can be configured for many purposes and environments.
- May be parts of even larger systems.
- Are in operation and used by several demanding and influential customers.

- Are primarily judged and valued by the presence of qualities such as speed, capacity, robustness, safety, and availability.
- Require hundreds of people for the development of a new release

In this chapter some of the experiences and practices used to build these large systems have been brought together into a development strategy named Integration Driven Development (IDD). Although IDD is intended for large systems, most of the practices are valid and can be applied also when developing systems that are not so large.

The General Idea

Iterations and Increments

The traditional way of developing systems, also known as the “waterfall” approach, can be roughly characterized as distributing the work to be done to various modules and components which are individually developed and tested. Toward the end of the project, these components are integrated and tested in a so called “big-bang” activity. Consequently, the system qualities can neither be seen nor guaranteed until late in the project. The commonly-agreed cure for the big-bang integrations is called “Incremental and Iterative Development”.

The single most time-consuming and difficult task when developing large systems is integrating the components developed into a complete working system. Regardless of the time spent in advance on carefully specifying interfaces and trying to predict the characteristics of the developed system, the integration will typically lead to unpleasant surprises: the time it actually takes to get the system up and running will certainly be longer than expected, the initial system behavior will not be as predicted, and the total cost for the final integration will exceed the wildest guesses.

IDD is a development strategy that focuses on early and frequent integration of small system changes to avoid the single “big-bang

integration" in the end. This is certainly the general idea of all modern iterative and incremental system development strategies. What makes IDD different is that it manages to apply the paradigm of incremental development to the development of large systems. This is done by:

- Focusing the incremental development paradigm on the top system level – the complete system – and also attempting to remove as many intermediate integration levels as possible.
- Recognizing that frequently doing new increments – for example each second week – requires development to be done *in parallel*. Sequential incremental development – having all teams working on the same increment - doesn't work for large systems.
- Recognizing that the management of parallel and incremental development requires careful planning, preferably based on anatomies.

The term "incremental and iterative development" is a bit confusing since both "iteration" and "increment" mean different things in different contexts. Large systems are developed iteratively and incrementally on many levels at the same time.

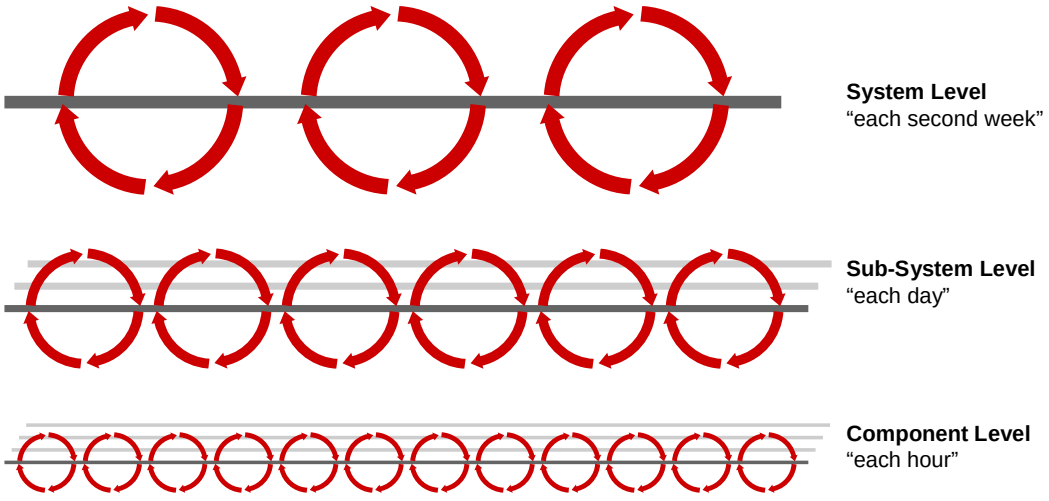

Figure 8.1 Three levels of integration and iteration frequency

A large system has several integration levels at which parts of the system are integrated into larger parts which are then tested (Figure 8.1). Each

integration level holds a specific test environment and a set of test methods to check the correctness of the integrated item. Furthermore, each type of item – component, subsystem, and system – can be developed iteratively with *different* iteration frequencies and *different* activities included in the iteration cycles.

In the same way, the term "increment" means different things to different people. Some say that an increment is the new version of an integrated item resulting from iteration. Others use the term increment to mean the activity needed to develop a new item version. That is, increment and iteration can mean the same thing. And still others say that the increment is the difference, in terms of functionality and characteristics, between two consecutive versions of an item.

So, most development organizations can argue that they do, and always have done, incremental and iterative development in some sense. The same organizations seldom produce new versions of working systems – at the top-most integration level – more often than twice a year. At least not when we speak about systems including thousands of components, some twenty or thirty subsystems, and hundreds of people involved in the development.

True incremental development is meant to be done at the top level of integration – the complete system level – and with IDD we strive for a new complete system version at least twice a month.

Working System Versions and Deltas

The vague terms increments and iterations are avoided when describing IDD. Instead IDD is based on the two fundamental concepts "Working System Version" and "Delta":

- Working System Version (WSV) – A WSV is a complete system version that has been demonstrated to work in at least one target-like environment without any major problems.

- Delta (Δ) – A delta is a system change, or more precisely, a specification of a system change, including the test cases or scenarios that must be passed to consider the change done.

The fulfillment criteria for allowing the quality stamp "working" on a WSV must be carefully defined for each system and system version. What "working" means differs among systems and organizations depending on the availability of test environments and the willingness for risk-taking. The reason for always having *working* system versions is to always have a stable design base for which quality is under full control: *We know exactly the things that work well, and the things that don't.*

A delta specifies a system change, but also the specific quality level for the resulting WSV. Different changes require different test activities to ensure a sufficiently good WSV.

In the reminder of this chapter we will sometimes use "step" to mean the same thing as a delta, and we will say that IDD is a strategy for stepwise development of large systems. Now, Figure 8.2 shows the general idea behind IDD and the stepwise development of systems.

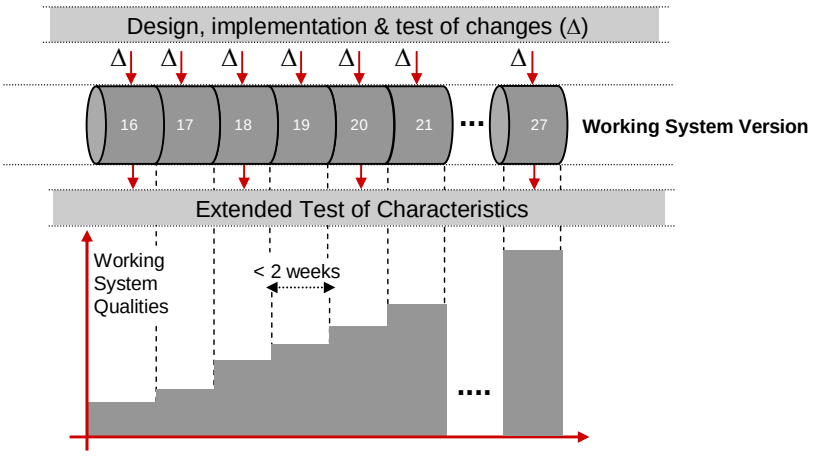

Figure 8.2: The General Idea

New working system qualities – functions and characteristics – are increasing stepwise with each new WSV, and a new WSV is produced each

second week. A new WSV is the result of adding or integrating a planned system change, a delta, to the previous WSV. That's the idea behind IDD.

The time interval between two consecutive WSVs is set by the average time it takes to integrate new and modified components with an existing WSV, do all the necessary tests and corrections to reach the quality level "working", and then make public a new WSV. Even for large systems this can be done in less than two weeks by, for example, having very small deltas, automating the regression tests, and having less strong and comprehensive criteria for "working". However, experience shows that each second week is a tough enough goal when we are required to "know exactly what works, and what doesn't".

It may be worth mentioning that the ambition with a WSV is *not* to have it shippable to a customer, or to put it directly into operation. The large systems we deal with can normally be configured in very many ways, and a WSV says something about only one or a few of these configurations. Furthermore, the time to ensure the expected level of robustness, capacity, scalability, safety, etc. is often several months for the official releases of these systems. The WSV should, however, be good enough to be used for *measuring* these critical characteristics. In that way it will be possible to check characteristics in parallel with the ongoing development of new WSVs.

Integration Driven Development

Overview

Figure 8.3 is an attempt to illustrate the key aspects of the IDD strategy. The picture is centered round the stepwise development of a system. New working versions of the system are produced every second week. Each WSV is developed by a cross-functional team – named Δ-teams – which has the end-to-end responsibility for its realization.

The typical size of a Δ-team is five to ten persons, and the development time may vary from a few weeks up to several months.

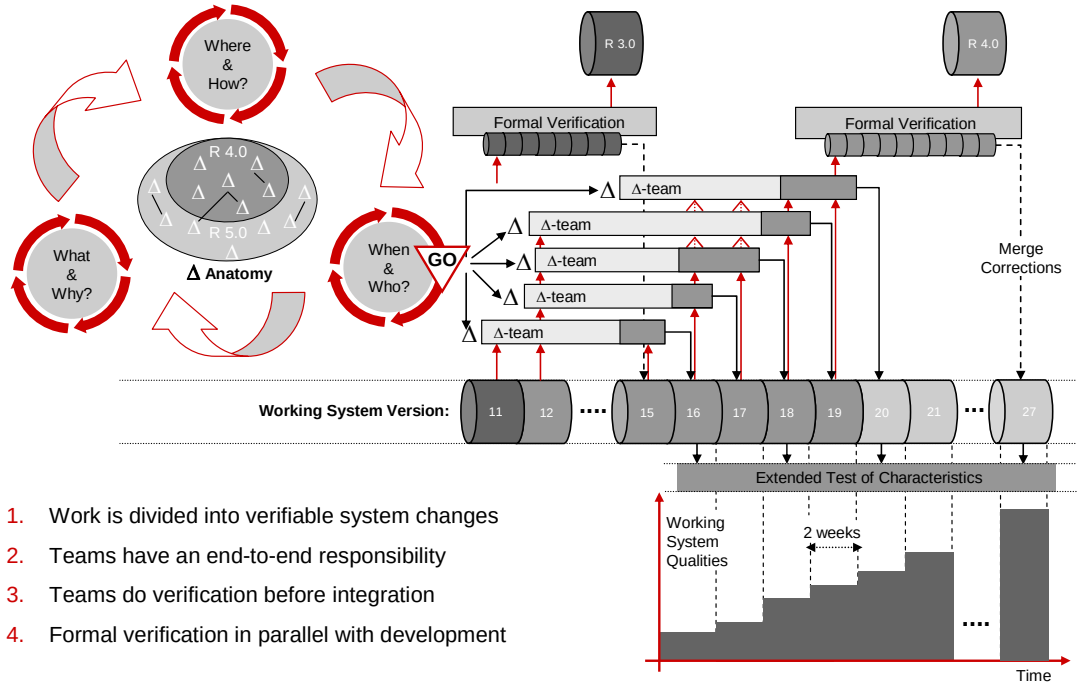

Figure 8.3: Integration Driven Development Overview

The task for a Δ–team is to design, implement, and test a specified system change – a delta. As can be seen in Figure 8.3, Δ-teams must work in parallel in order to keep up the bi-weekly pace of new WSVs. To avoid a situation of having several teams working with the same software components in parallel, dependencies between deltas must be clearly understood, and the order of realization carefully planned.

Management of system development is organised around the Δ-anatomy which shows currently-planned system releases, their content in terms of deltas, and dependencies among the deltas constraining their order of realization. The management part of IDD is illustrated in the upper left corner of the picture and is divided into three continuously-ongoing activities:

- "What and Why?" – The purpose of this activity is to specify the content of system *releases* – official system versions delivered to customers – in terms of new functions and capabilities.

- "Where and How?" – This is about transforming the new functions and capabilities into small, verifiable system changes (deltas), and clarifying their dependencies in a Δ-anatomy. This activity includes specifying each delta in terms of affected components and interfaces, and roughly estimating the size of the delta.
- "When and Who?" – This activity means scheduling deltas in a development plan which shows their order of realization and integration. The integration order is determined by the dependencies between deltas and the availability of developers. The activity includes "kicking off" the realization of deltas by coming to an agreement with Δ-teams ready for new tasks.

Before a Δ-team is allowed to deliver the realization of a delta, they must make sure that their new and modified components work in a complete system version, a version based on the latest WSV. That is, Δ-teams don't deliver components or subsystems but always new and complete system versions that have been demonstrated to work in a target-like environment. This means that once a WSV is sent to "Extended Test of Characteristics" or "Formal Verification" we don't expect to find any trivial software faults.

Note that formal verification is *not* done for each WSV but only for system versions representing the realization of a complete system release (for example, WSV 19 in Figure 8.3). Faults found during formal verification are corrected immediately in a separate verification track. Needless to say, these corrections must, sooner or later, also be merged into the main development track – the WSV track.

IDD embraces four key development principles listed in the left bottom corner of Figure 8.3. In the following sections we will dig into these a little bit deeper.

Work Is Divided into Verifiable System Changes

The first development principle is: *The scope for a system release is divided into small and verifiable system changes, instead of being refined in detail and distributed onto subsystems and components.*

The common way of starting development work is to capture and specify the requirements stakeholders have on the system release, and then refine and allocate detailed requirements onto the subsystems and components that make up the hierarchical system structure. After that, people are asked to implement the refined requirements in the components they own.

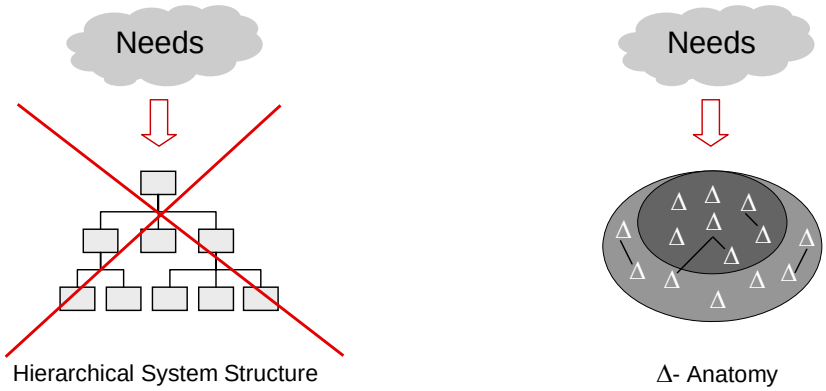

Figure 8.4: Needs are divided into verifiable deltas

This approach of detailing and allocating requirements down to the lowest level of components comes from the “waterfall strategy” and the belief that it is both necessary and possible to specify everything in detail before the implementation can start. And, of course, this approach must lead to a big bang integration in the end. It will not be possible to test the system, not even to build a running system version, until each and every component has been correctly implemented, delivered, and integrated.

First, before we take a look at system changes and deltas, let’s be clear about requirements. In IDD we try to avoid entirely the word “requirement” since its very existence tends to generate an overwhelming “requirements management process” with no or little value. Instead, we use the word “needs” for the needs stakeholders have on the top system level. That is,

needs express desired system qualities – new functions and characteristics – only visible and of interest when considering a complete system version. The rest of the information describing the things developed is called “design information”: information describing the use and construction of existing and planned system versions.

The idea behind the first development principle is to transform stakeholders’ needs into a set of system changes that we call deltas. Stepwise implementing these deltas will in the end result in a new system release fulfilling the needs of the stakeholders. The key qualities we want deltas to have are the following:

- The realization and integration of a delta must always result in a new working system version.
- The description of the delta should include an identification of new and affected components and interfaces, together with the system test scenarios which will be used to verify an acceptable realization of the delta.
- Each delta should be minimal in the sense that it can not be divided into parts which, in turn, could serve as deltas.

Regardless of the way you choose to transform the needs into deltas, there will probably be deltas that depend on each other in different ways. We use the Δ-anatomy to visualize, communicate, and keep track of these dependencies (Figure 8.5).

The Δ-anatomy differs significantly from the system anatomy described elsewhere in this book, both in terms of content and usage. The elements in a system anatomy are the system’s key capabilities, whereas the elements in a Δ-anatomy are the currently planned changes needed to obtain some new system capabilities. Another important difference is that a system anatomy is rather stable compared with a Δ-anatomy. The Δ-anatomy constantly undergoes change. New needs and insights appear, deltas will be added, removed, and redefined, and there will typically be a new Δ-anatomy every week.

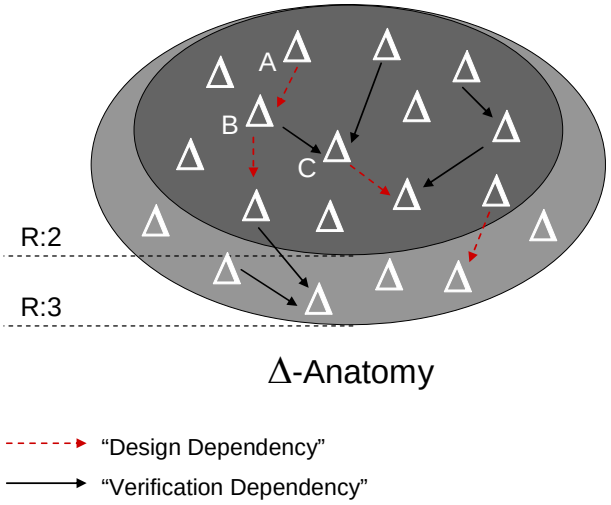

Figure 8.5: The Δ-Anatomy

Figure 8.5 shows the key elements of a Δ-anatomy: releases, deltas, and dependencies among deltas. The next release, R:2, currently contains thirteen deltas whereof four of them can be implemented and integrated independently of other deltas. The dependencies constrain the order in which the deltas can be done, and determine the possible degree of parallelism. There are two types of dependencies that should be considered and managed in a Δ-anatomy:

- "Design Dependency" – In Figure 8.5: ΔA should be completed before the *design* of ΔB can start. The motivation for controlling this kind of dependency is to avoid, as far as possible, parallel development of software components, and by that, avoid the costly task of merging code.
- "Verification Dependency" – In Figure 8.5: ΔB should be completed before *verification* of ΔC can start. This dependency indicates that design of ΔC may start before ΔB is ready, but verification of ΔC can only be done after the completion and integration of ΔB.

If you find it difficult or unnecessary to distinguish between these two dependencies you can combine them into one general dependency or

relation named “should be done before”: ΔA *should be done before* ΔB, and ΔB *should be done before* ΔC.

Teams Have an End-to-End Responsibility

The second development principle is: *We have cross-functional teams with an end-to-end responsibility for the realization of a new system version, instead of specialist teams delivering parts to each other*.

True teamwork is essential in IDD. Having stable, well-functioning teams may be the single most important key to successful systems development. However, good team work requires careful staffing, training, and encouragement. It won’t happen just by assigning people a common task.

Having good teams is eased by a reasonable team size (7 ± 2 persons), co-location and team rooms, full time members, long-lived teams, and by giving teams the opportunity for making a true commitment, that is, giving them time for analysis, planning, and negotiation.

In the original IDD strategy we promote stable, long-lasting teams that can take on the realization of an arbitrary delta. In practice, for large systems, it’s never an arbitrary delta. The teams will be specialized in deltas affecting specific parts of the system. There will also be deltas that require new team constellations. This must be dealt with when planning the development and integration order.

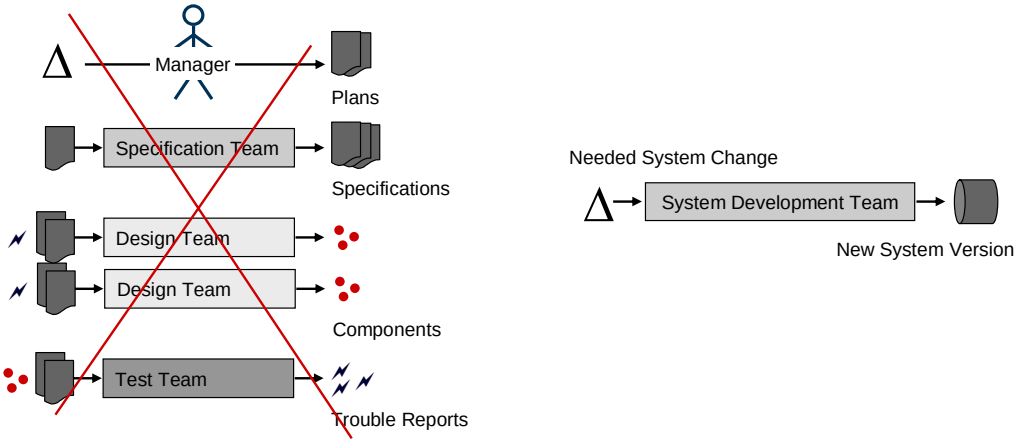

Figure 8.6: Teams have an end-to-end responsibility

Figure 8.6 illustrates the intention behind the second development principle and the formation of Δ-teams.

Large organizations tend to form teams and competences around specific development activities. There may be a “planner” who plans the work, a “specification team” specifying conceptual solutions and doing overall design, “design teams” doing detailed design and delivering components, and finally, “test teams” who integrate components, run test cases, and deliver trouble reports. The many handovers this leads to have been shown to be rather inefficient. However, the striking thing is that there is seldom a team that actually has the explicit task of delivering the new, complete and working, system version.

A Δ-team includes all the competences needed to develop a system version; for example, planning and tracking, system design, software design, hardware design, configuration management, testing, and documentation. Each member of the Δ-team has the role of “system developer” and the team has not finished its job until it has delivered a new working system version. Hence, every Δ-team is a “system development team” in a literal sense.

How the development work is done is decided by the Δ-team. IDD does not prescribe any methods or tools to be used, and teams are encouraged to use whatever practices they prefer. This includes agile practices such as SCRUM and “sprints” for development planning, test-driven development, pair programming, daily build, and refactoring.

There is, however, one agile practice that needs to be applied with extra care: time boxing. In the agile development method SCRUM, teams do their work by running subsequent sprints with a fixed time length, for example, two weeks. Every second week the team delivers a result that is evaluated by stakeholders. In contrast, a Δ-team is *never* allowed to deliver if things don’t work as agreed, even when the scheduled delivery date will be passed. We never accept non-working WSVs. In IDD, teams practice “quality boxing” rather than time boxing.

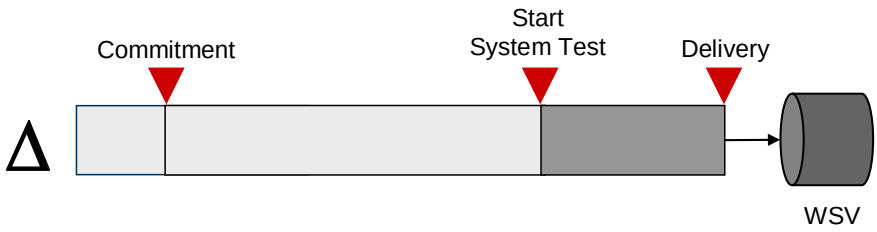

Figure 8.7: Three mandatory milestones

There are three milestones that each Δ-team must pass (Figure 8.7):

- "Commitment" – The team has reviewed and analyzed the information specifying the delta, detailed the tests and quality assurance activities that must be done before delivery, clarified how the change will be implemented, and agreed on a delivery date.
- "Start System Test" – All new and modified components have been checked as agreed upon in the "Commitment", and the team is now ready to start formal testing activities with a system build based on the latest WSV. These test activities should not exceed the maximum time span decided upon for two consecutive WSVs (for example, two weeks). Throughout this test period there can be no new WSVs delivered from other teams.
- "Delivery" – All quality assurance activities agreed upon at "Commitment" have been carried out, faults have been corrected, and no major problems remain. The quality mark "working" can be stamped on the team's final system version, and the team can deliver a new official WSV. Regression testing – to assure that things that worked in the previous WSV still work – is always included as a mandatory part of the system test.

The fulfillment of milestone criteria is checked by the "When & Who"-team, which plans and tracks system development (see Figure 8.3).

Before ending the discussion about Δ-teams, it may be in place to sort out a general misunderstanding about Δ-teams and their relationship to Development Management concerning the three activities What & Why,

Where & How, and When & Who, found in the upper left corner in Figure 8.3. The team running the "Where & How" activity – specifying and maintaining the Δ-anatomy – is staffed with members from the Δ-teams. There is no such thing as a "hand over" from an "anatomy team" deciding what to do, to a Δ-team doing what's been decided. Structuring and specifying deltas, and sorting out their dependencies, must be done by persons that fully understand how the system is constructed; that is, by system developers.

Teams Do Verification Before Integration

The third development principle is: *Teams deliver complete and working system versions, instead of insufficiently-tested components with unknown effects on the system.*

If we want teams to deliver components that work, we must provide them an opportunity to check those components in the context of the latest working system version. This is what the third development principle is about.

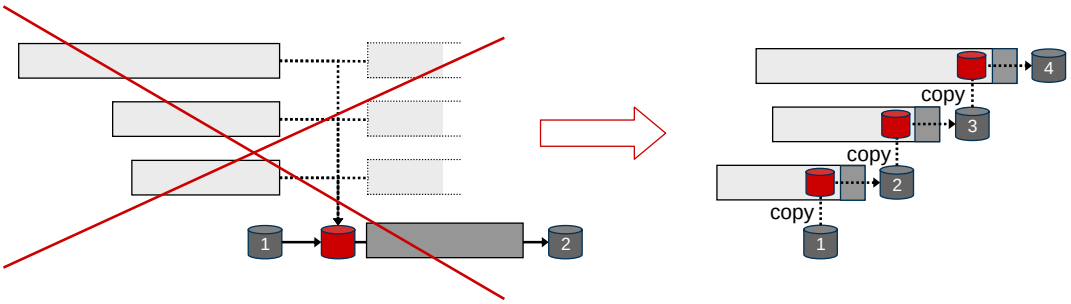

Figure 8.8: Teams do verification before integration

The situation we want to avoid, the left part of Figure 8.8, can be described as follows. We have a working system version (Version 1) and it's time for the three development teams to deliver their components. When integrating the components with the working system version, we get a non-working system. The non-working system is handed over to an "integration and test team" which is supposed to make the system work. At the same time, the

three development teams rush on to the next increment, basing their development on the defective components that have been delivered. The "integration and test team" need corrected components, but don't allow any other additions or modifications to the components. Hence, the development teams must keep at least two tracks for each component, and correct each fault twice. In short, we get a messy and inefficient situation.

If we instead provide each team with a complete system test environment where they, by themselves, can make sure that their components work *before* they are delivered and "integrated", we reduce a lot of waste.

The key to having "verification before integration" is to forbid development directly on the WSV branch. Development work is done on separate branches isolated from the WSV branch. To get this to work, you need to invest some time and money in a good software build and configuration management system. However, the general idea is straight forward, as outlined in Figure 8.9.

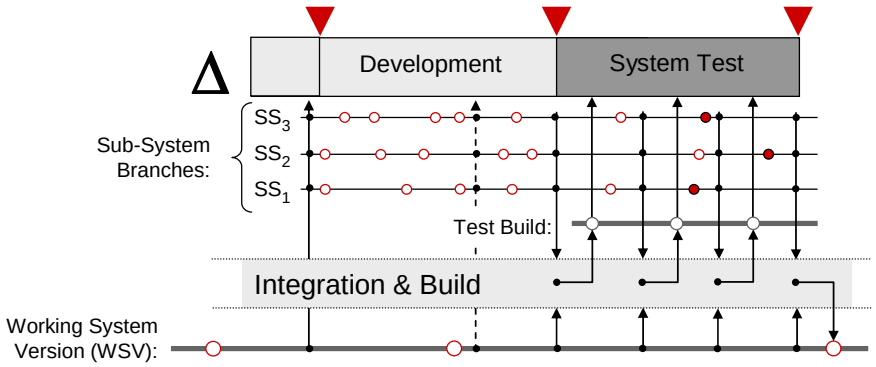

Figure 8.9: Δ-team with subsystem branches

For complex software configurations it can sometimes be a good idea to centralize the creation of branches and software builds to a specific team serving all Δ-teams with "Integration & Build". However, with a fast and well-functioning build and configurations management system, it is still preferable to have the Δ-teams creating branches and builds by themselves. With many ongoing teams, quick build procedures, and many target test environments available, there will undoubtedly be a need for many builds.

Consider a Δ-team with the job of implementing a delta affecting three subsystems: SS1, SS2, and SS3 (Figure 8.9). Three development branches are created from the subsystem versions in the latest WSV, and the team starts their development. In Figure 8.9 new subsystem versions are shown as small circles on the branches. After a while there is a new WSV available. If the WSV contains a new version of any of the three subsystems, the team must do a "rebase" and merge the new subsystem version with their own latest version. This will result in a new version on the corresponding subsystem branch. The team can, of course, as often as they like and until it is time to deliver, create system builds based on the latest WSV to check how their modified components work in the current system context. When the phase "System Test" starts, the team is guaranteed that there will be no new WSVs until they deliver their own WSV.

Formal Verification in Parallel with Development

The fourth and last development principle is: *We do formal verification on a specific "verification track" in parallel with continued development, instead of halting development, and use the development track for verification and corrections.*

Developing and maintaining large systems requires, in most cases, several types of "tracks". A track is a sequence of system versions with a specific quality or purpose which we want to, or must, separate from other sequences.

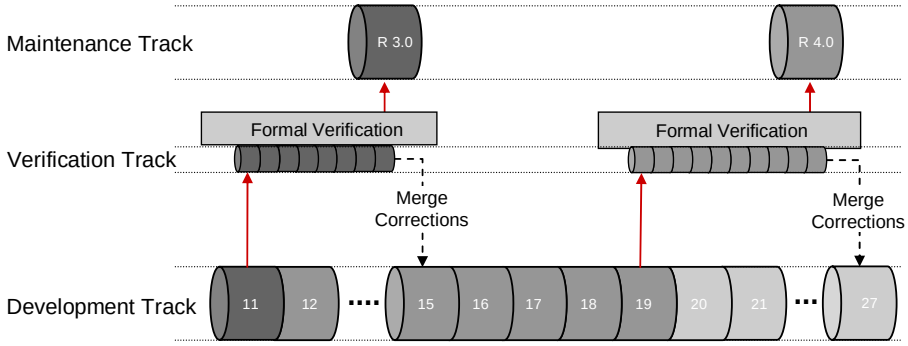

Figure 8.10: Three types of tracks

Within the IDD strategy we define and use three types of tracks as illustrated in Figure 8.10.

The “development track”, sometimes referred to as the “main track”, is the one single official sequence of system versions we use for growth of functionality and characteristics. Within the IDD strategy, the development track is the sequence of WSVs.

For large systems formal verification often implies integrating and verifying the system with other large systems and checking several possible configurations of these systems. Therefore, formal verification normally takes several months, a time period during which we need to proceed with development.

The “verification track” is used only to correct faults found in the system, or, in other words, the verification track is a sequence of system versions where we only allow growth of quality. Formal verification ends when the quality of the system is considered to be good enough for a new system release. The verification track is closed when verification ends and, at that point, the latest system version in the verification track is the new system release. What remains to be done is ensuring that all corrections made in the verification track are also made in the development track.

Each system release will result in a “maintenance track” of its own. As with the verification track, the maintenance track is only used for quality growth. The maintenance track is closed when all customers and users of that release have upgraded their system to a more recent release. The big challenge for all system providers is to keep the number of releases being

maintained to a minimum. The maintenance cost for a system provider is directly related to the current number of maintenance tracks.

To complicate this picture even further, each system release may be adapted towards the needs of a specific customer. This means that every release may result in several customer-specific tracks which need to be maintained, and sometimes even evolved, independently. But that's another story, perhaps in another book.

Variants

Subsystem Teams

In core IDD we have Δ-teams (Figure 8.3). Δ-teams are expected to be long-lasting, cross-functional teams that can take on the realization of any delta regardless of which subsystems and components that delta will affect. This requires the ownership of code and design to be everyone's responsibility; that is, each Δ-team has the right and ability to do all the changes needed to implement the current delta. In terms of responsibilities, the Δ-team is similar to so-called "feature teams", although a delta is seldom a complete system feature or capability.

For large, complex, and technical systems, clear individual ownership for each and every component is considered crucial in order to preserve the architecture and design idea over time. Moreover, components normally exist in a large number of versions and variants being maintained. Changing or correcting a component requires a deep understanding of the dependencies among these versions and variants.

Thus, for some systems and organizations, it is more common to form teams and responsibilities around components and subsystems than having general development teams that are expected to do work everywhere in the system.

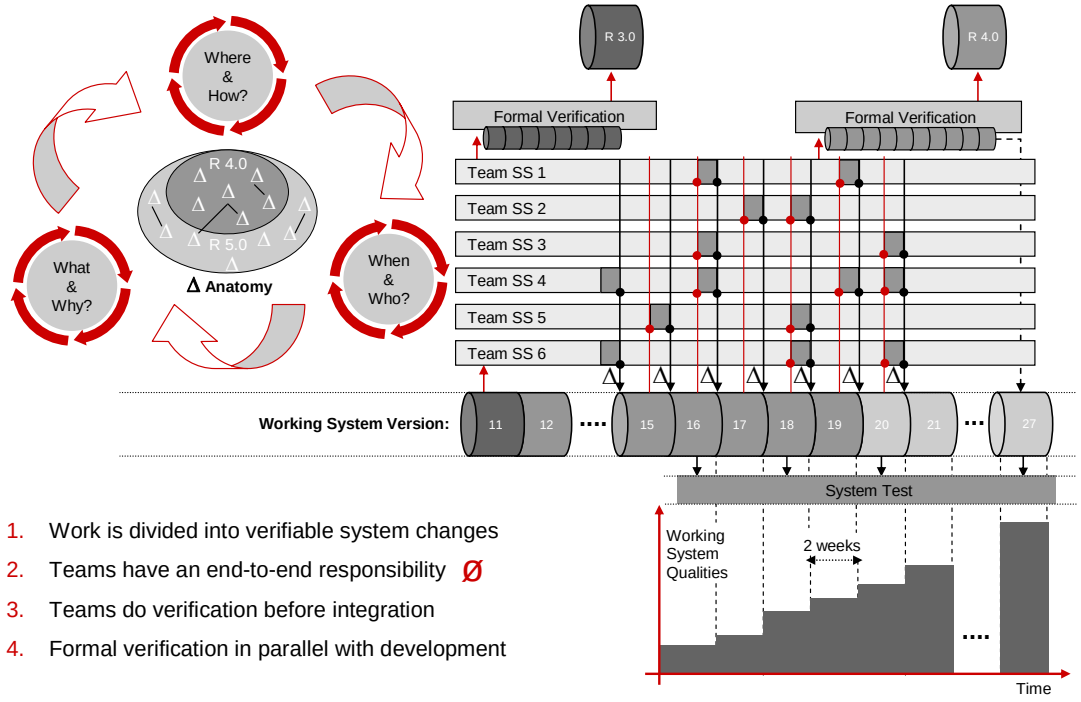

Figure 8.11: Subsystem Teams

In the "Subsystem Team" variant of IDD, we replace Δ-teams with subsystem teams (ss-teams) which develop and deliver new versions of the subsystems they own (Figure 8.11).

In this variant, the second IDD principle, "Teams have an end-to-end responsibility", is not fully applied. The ss-team is of course cross-functional (cross-discipline), but, since a delta seldom affects only one subsystem, the realization of a delta becomes a responsibility shared among several teams. Still, an ss-team always delivers a WSV, not just a "good looking" subsystem.

The second development principle is the only one you need to compromise when you choose ss-teams. For example, the third principle, "verification before integration", is still applied. In Figure 8.11, the delta resulting in WSV 17 impacts the subsystems SS1, SS3, and SS4. Before the corresponding ss-teams can deliver their modified subsystems, they must complete a common test to show that their subsystems work together; that is, they must show that the new system version works and that the delta is properly implemented.

As discussed earlier, in most cases a Δ-team works with several software branches, one for each subsystem being changed (Figure 8.9). An ss-team will work with one and only one subsystem, but will normally need to work with several deltas in parallel (Figure 8.12).

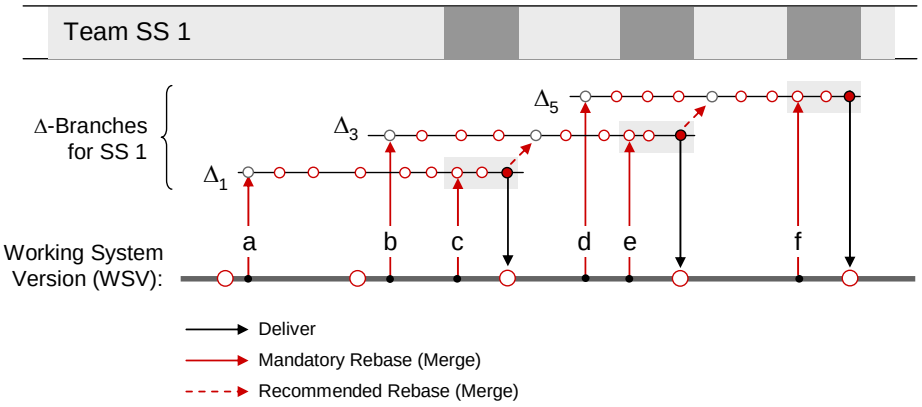

Figure 8.12: Δ-branches for a subsystem

Figure 8.12 shows how an ss-team sometimes needs to work with two deltas in parallel. This is done by creating a new subsystem branch each time the realization of a delta begins. The branching is done from the latest *working* subsystem version. Note that a "mandatory rebase" becomes trivial (unnecessary) if the previous "recommended rebase" is done. On the other hand, the mandatory rebase marked "e" may imply code merge if the previous recommended rebase was not done.

Frequent System Build

The variant "Frequent Build" can be viewed as the well-known software development concept "Daily Build" applied to a large software system; perhaps not daily, but frequently (Figure 8.13). In contrast to core IDD and the subsystem variant, this variant may be an alternative only when considering large *software* systems.

With Frequent Build we remove the subsystem integration level and tests are focused directly on the system level. This also means that developers and teams don't deliver subsystems, but components. The team concept in

this variant is not clearly defined. Or, more precisely, we don't prescribe how teams should be formed. Teams may be formed around groups of components or temporarily around features; these different types of teams coexist and a developer may be a member of several teams. Moreover, you can choose to have developers doing a system test or have a specialized team for it. In short, we don't apply the second principle, "teams have an end-to-end responsibility".

However, the first development principle is valid and development is managed as in core IDD. The development scope is divided into deltas and the integration plan shows when deltas are expected to be integrated and ready for testing.

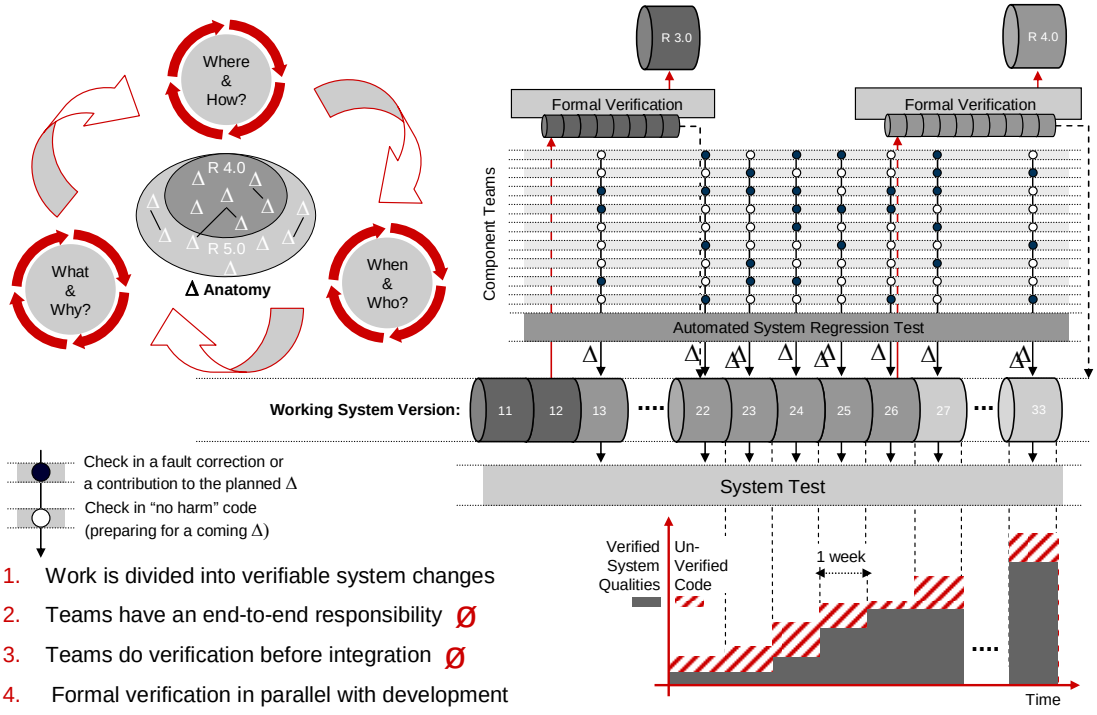

Figure 8.13: Frequent System Build

The WSV is a baseline created weekly in the one and only code branch. Every night a complete system build is done, based on the latest "checked in" component versions, and an automated system regression test is run. The quality and coverage of the regression test determines the quality level of the WSV. In the best of worlds, the regression test will check that basic functions which worked in the latest *released* WSV are still working (for

example, WSV 12 in Figure 8.13). So, when establishing a WSV baseline we know that basic functions still work. However, a check remains to see whether the newly integrated deltas work. This is done by a System Test in Figure 8.13. Hence, the system test is done *after* integration with WSV and, obviously, the third development principle isn't applied. This also means that the quality stamp "working" is weaker than in the IDD variants described earlier.

As mentioned earlier, applying the third principle, "verification before integration", implies branching and *no* development directly in the WSV-branch. The main advantages with this approach are:

1 It's possible to always have full control of the WSV content and quality.
2 It's possible to halt the realization of a delta without the cumbersome work of removing things from the WSV.
3 Teams have a stable, high-quality test base (the latest WSV) and are not forced to always use the latest "checked in" code for testing activities.

In which situations can we still be motivated to consider the Frequent Build variant? The short answer is: When merging code and models is considered to be a nightmare, and you have no idea how to avoid it. Having teams working on their own development branches requires a modular system architecture, clear interfaces, and components that implement well-defined parts of the services provided by the system. If, from a total system perspective, every verifiable change requires changes to a majority of the components, then developing many changes in parallel on different branches may not be the best approach. Instead, it might be better to strive for one single development branch and avoid the risk of spending most of the development time merging versions developed in parallel.

Single Track

The third and last variant is called "Single Track". This variant is very similar to the previous Frequent Build, with the only difference being that

not even the forth IDD principle is applied. And, when only using the first development principle, it is indeed a bit difficult to argue that Single Track is an IDD variant. The reason for bringing up Single Track is that it tackles one of the most urgent issues for providers of large, evolving, multi-customer systems, namely, cutting down the number of versions maintained. One system version means one and only one version of each component, and consequently, changes and corrections are done only once.

The idea behind Single Track is to have, from a system-provider perspective, one single version of the system at every moment in time. The only existing sequence of system versions – the one single track – must serve for new system releases, maintenance releases, and for development.

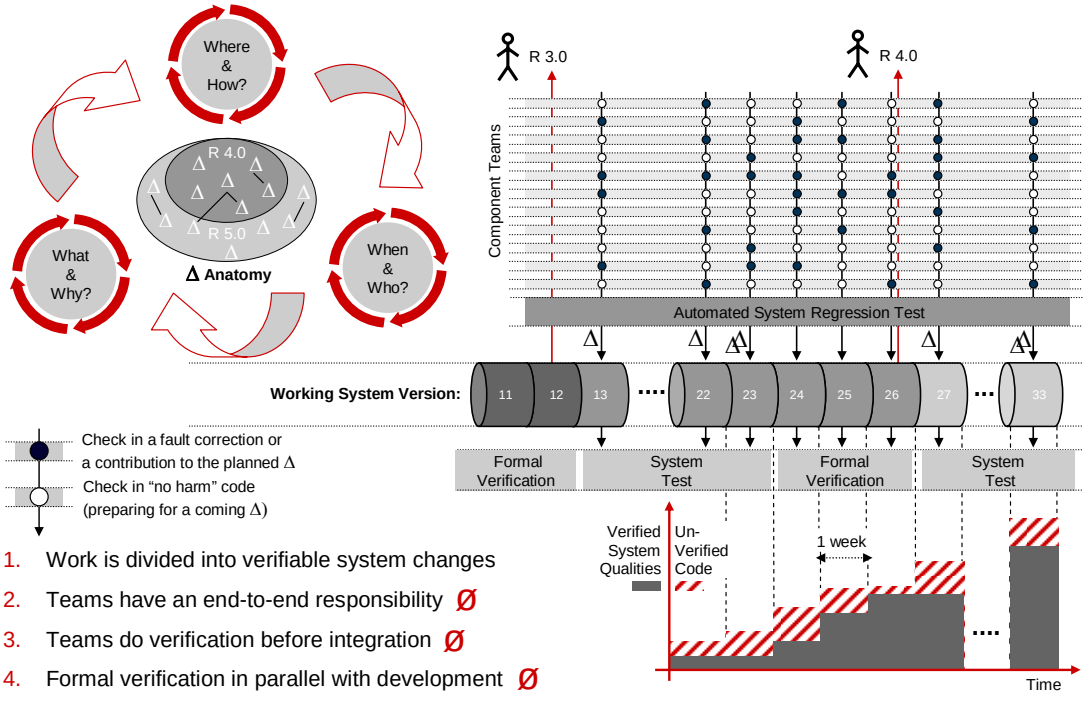

Figure 8.14: Single Track

In Figure 8.14 WSV 12 and 26 are released directly to customers. There is no verification track, and no maintenance tracks for corrections on released system versions. There is one and only one development track: The WSV-track. In contrast to core IDD, the quality of the WSV varies over time. All through Formal Verification development is halted and new WSVs are used to correct faults and improve quality (WSV 24, 25, and 26 in Figure 8.14).

Throughout the development phase, which also means the system testing, the latest WSV is the only existing design and test base. Therefore, we must allow developers to check in "not ready code" used to test and prepare for coming deltas. The key task for the automated regression test is to ensure that the "not ready code" doesn't break the system, and that everything still works as expected. The amount of "no harm code" in the WSV will vary over time, but most likely there will always be some code that doesn't contribute to the system's working functions and qualities.

Whether a "single track strategy" is a success or not isn't primarily about how well the development method is established; it is about the current business model and the relationship with customers. Here are some issues that need to be sorted out before trying to establish Single Track:

- Will customers accept that maintenance releases are replaced with ordinary scheduled releases?
- Will customers accept the risk of replacing a defective version with a corrected one when the new version also includes features they haven't asked for? What about the existence of "no harm code"?
- How do we ensure that customers don't use the new features they get for free when sending them corrections included in an ordinary release?
- For how long can we guarantee backwards compatibility? What about maintenance after that? Can we "force" customers to upgrade?

Let's end this section by concluding that IDD and Single Track are two different solutions for two different types of waste. The IDD strategy is primarily an attempt to solve the Quality Problem: *We find far too many costly faults and unwanted characteristics far too late*. Whereas Single Track focuses on the Track Problem: *We have far too many system and component versions that we develop and maintain in parallel*. The choice of strategy depends on the problem you want to solve. Neither will solve both.

Summary and Discussion

Agile methods emerged as a well-justified reaction to the heavyweight development ideals promoted by formal systems engineering and the waterfall approach. As often happens with such paradigm shifts, the pendulum may swing all the way to the other end of the spectrum. Agile methods are sometimes seen as the “silver bullet” that will solve software development issues once and for all. However, with the increasing scale of projects, concerns with “agile-only” methods begin to show up. IDD might be seen as yet another step in the evolution of methods, combining careful planning with agile development. In a sense, the IDD approach is an empirical, strong-voiced answer to the issue of whether plan-driven or agile methods should be used. The answer to which is – both. With the increasing scale of system development, an element of stability in agile methods is indispensible.

A critical element in the IDD strategy is the Δ-anatomy. The anatomy construct has remained in place at Ericsson in one form or another through all the years since it was first conceived almost two decades ago. The main reason for this is undoubtedly that it visualizes the most important factors when developing complex systems: how system capabilities or changes depend on each other.

Moreover, an anatomy is consciously kept as simple as possible in order that it can be easily understood by all stakeholders. The importance of shared or common understanding cannot be overestimated. If there is no common view of the development scope, the risk of failure is high.

Concluding the chapter, the main point about IDD can be expressed as follows. If there are many agile teams working at the same time, and all teams develop parts of the same system, then it is necessary to spend time on coordinating these teams. This effort primarily involves dividing the development scope into suitable system changes, and deciding their best order of integration. Having done that, it’s possible to have many teams working in parallel, letting them apply whatever agile practices they like, and still having them stepwise develop one and the same system. Therefore,

combining agile development practices with plan-driven methods is not only possible, it’s probably necessary in order to improve large scale development performance.