Agile and Incremental Development of Large Systems

Lars Taxén and Ulrik Pettersson

Linköping University Post Print

N.B.: When citing this work, cite the original article.

Original Publication:

Lars Taxén and Ulrik Pettersson, Agile and Incremental Development of Large Systems, 2010, 7th European Systems Engineering Conference (EuSEC 2010), Stockholm, Sweden, May 23–26.

Postprint available at: Linköping University Electronic Press
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-111349

Agile and Incremental Development of Large Systems

Lars Taxén
Linköping University
lars.taxen@telia.com

Ulrik Pettersson
SAAB Aerosystems
ulrik.pettersson@saabgroup.com

Copyright © 2010 by Lars Taxén, Ulrik Pettersson. Published and used by INCOSE with permission.

Abstract. Agile methods have been suggested as a means to meet the challenges in large systems' development, in particular large-scale development of software. However, with increasing scale, limitations of agile methods begin to materialize. To this end, we outline an integration driven development (IDD) approach that combines plan-driven, incremental development with agile methods. The planning is based on a visualization of dependencies between needed system changes – deltas – called the *anatomy*. The IDD approach originated in the Ericsson telecom practice more than a decade ago, and is now used regularly in large system development projects.

Introduction 1
Background 2
Integration Driven Development 3
The anatomy 3
Planning 6
Realization 7
Variants 8
Variant 1 - Sub-system teams 9
Variant 2 - "Daily build" 10
Discussion and Conclusions 11
References 12

Introduction

The development of large systems implemented in hardware and software pose immense challenges. In particular, large software projects often result in failures or less than anticipated results (e.g. Standish Group International 2009). The trend towards increasing size and complexity of systems shows no signs of flattening out. On the contrary, ultra-large-scale (ULS) systems are expected in the near future (SEI Carnegie Mellon 2006).

ULS systems "will push far beyond the size of today's systems and systems of systems by every measure: number of lines of code; number of people employing the system for different purposes; amount of data stored, accessed, manipulated, and refined; number of connections and interdependencies among software components; and number of hardware elements. The sheer scale of ULS systems will change everything" (ibid, ix). In particular, the social aspects need to be considered:

> We will need to look at [ULS systems] differently, not just as systems or systems of systems, but as socio-technical ecosystems. We will face fundamental challenges in the design and evolution, orchestration and control, and monitoring and assessment of ULS systems. These challenges require breakthrough research. (ibid, ix)

In this contribution, we outline an integration driven development (IDD) approach towards the development of large systems, which combines plan-driven, incremental development with agile methods. The planning is based on a visualization of dependencies between needed system changes – deltas (Δ) – called the *anatomy*. IDD originated in the early 1990s at Ericsson, a major provider of telecommunication systems and services worldwide. During the years the approach has been substantially elaborated, and is now used regularly in large system development projects. Thus, we propose that the principles inherent in, and the experiences from using the IDD approach, may be valuable for addressing the challenges in large systems' development.

Background

The traditional way of developing systems, also known as the "waterfall" approach, can roughly be characterized as distributing the work to be done to various modules that are individually developed and tested. Towards the end of the project, these modules are integrated and tested in a so called "big-bang" activity (see Figure 1). Consequently, the system qualities cannot be guaranteed until late in the project. This approach has a number of reported problems (see e.g. Royce 1970; Karlsson 2002; McConnell 2004) – requirements may change during the implementation; implementation problems may be discovered late in the project; lack of user involvement after the specification is written; testing resources such as test equipments and experienced testers are unevenly utilized towards the end of the project; to mention but a few.

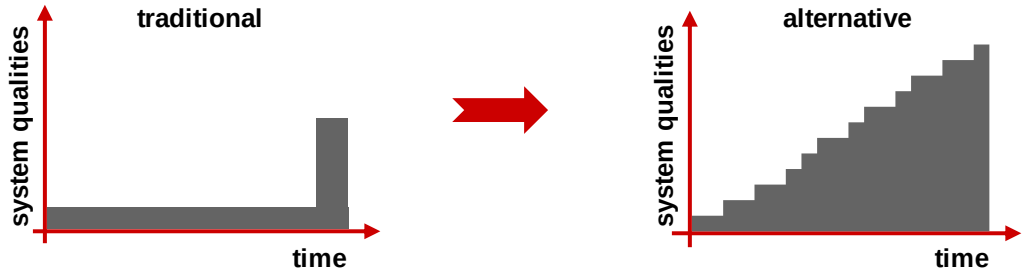

Figure 1: The general idea

The alternative way is to develop the system in many small steps – Δ:s – and verify the system after each step. This means that each step:

- is a realization of a *planned* system change (Δ),
- results in a *working* system version ready for advanced tests in target environment,
- is made within a few weeks or even faster.

There are several generally agreed advantages of developing a system in small steps:

- *Continuous quality assurance* – frequent and relevant feedback improves quality.
- *Reducing risk* – by reducing the size of a step we reduce the time and money invested in the step, and hence, the risks attached to it.
- *Tracking progress* – small steps provide frequent and indisputable milestones since true progress is visible at the end of each step.
- *Improving performance* – the things that you need to do frequently are things that you

become good at and do efficiently.

- *Increasing flexibility* – we are never more than a few steps away from formal verification and a possible delivery to the customer.

Agile methods such as "daily build", Scrum, XP (Extreme Programming), continuous integration, Evo (Gilb 1989) and others, are examples of stepwise development approaches (see e.g. Meso and Jain 2006, for a comprehensive overview). No doubt, agile methods have proven viable in industrial development of large systems (Larman and Vodde 2008). However, with increasing scale of applications, issues begin to materialize:

> In recent agile methods workshops with our large-company industry affiliates, the participants have unanimously agreed that agile methods have helped them become more flexible and adaptive to change. But they have also agreed that scalability and legacy practices have limited their range of adoption of agile methods. (Boehm, in Fraser et al. 2006, 226)

Scalability issues are, among other things, "team-of-teams coordination and change management, independent–team product interoperability, multi-customer change coordination and unscalable COTS[1] or architectural sub optimization on early increments" (ibid).

A central issue for scaling agile methods is to strike a balance between agility and commonality:

> Being successful in large-scale product development requires finding ways to enable self-directing teams to work towards a common goal without compromising empowerment and feeling of ownership in the teams - a task easier said than done. (Vilkki, in Fraser et al. 2006, 228)

We claim that the IDD approach is one way to address scaling of agile methods and other issues relevant for large scale system development. The paper is organized as follows. In the next section, we outline IDD in three steps: we describe the system anatomy, the planning process, and the agile realization of planned changes (Δ). This is followed by two variants of the basic IDD strategy: one with focus on sub-system teams, and one variant similar to the well known "daily build" concept for software development. Finally, we discuss the IDD strategy and draw some conclusions.

Integration Driven Development

Integration Driven Development consists of two major activities: rigorous anatomy-based planning and agile realization of planned changes done by "Δ teams" working in parallel. We begin with a short description of the anatomy construct.

The anatomy

The anatomy is the key to master parallel, stepwise development. It is an illustration – preferably on one page – that shows *the dependencies between capabilities* in the system from start-up to an operational system (Adler 1999; Anderstedt et al. 2002; Jönsson 2006; Taxén and Lilliesköld 2005). Here, "capability" shall be understood as the capability of a certain system element to provide a utility that other system elements need.

An example of a realistic system anatomy from the Ericsson practice is shown in Figure 2:

[1] Commercial, off-the-shelf (COTS) is a term for software or hardware products that are ready-made and available for sale, lease, or license to the general public.

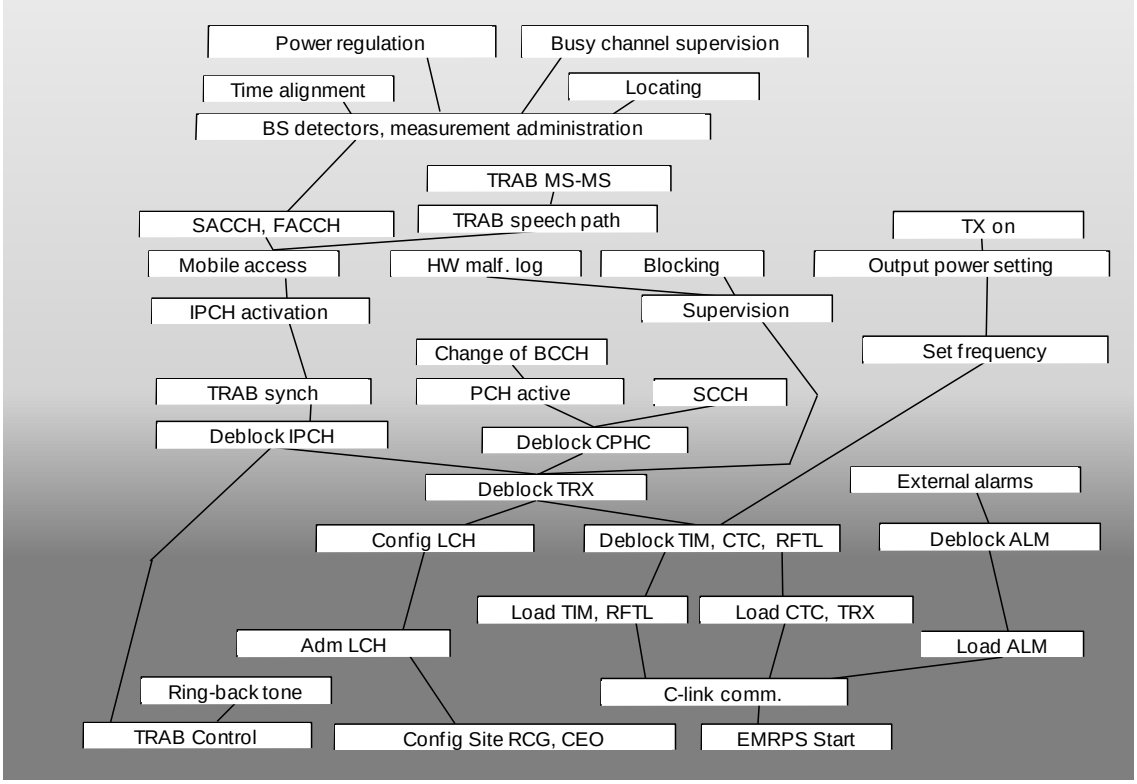

Figure 2: An anatomy for a radio-base station

Each box (the details of which are less important here) should be read as a capability provided by one or several modules (subdued in the figure). The dependencies (lines) proceed from the bottom to the top of the anatomy. For example, the capability "EMRPS Start" (an extension module regional processor with a speech bus interface) is a basic capability. If this capability fails, the whole system will fail. In line with this, the gist of the anatomy-centric approach is to design and test the system *in the same order as the capabilities* are invoked. In a metaphorical sense, this can be seen as the order in which the system "comes alive"; hence the term "anatomy".

When doing IDD the anatomy is a vital "input" in the creation of a slightly different type of anatomy – the *Δ anatomy*. The Δ anatomy does not show all the capabilities of a system, but only the changes and addition of capabilities currently planned for realization (see Figure 3). The white squares represent Δ:s in various stages of completion. For simplicity of drawing, the Δ anatomy is constructed to be read from the top down in contrast to the anatomy in Figure 2, which is read bottom up.

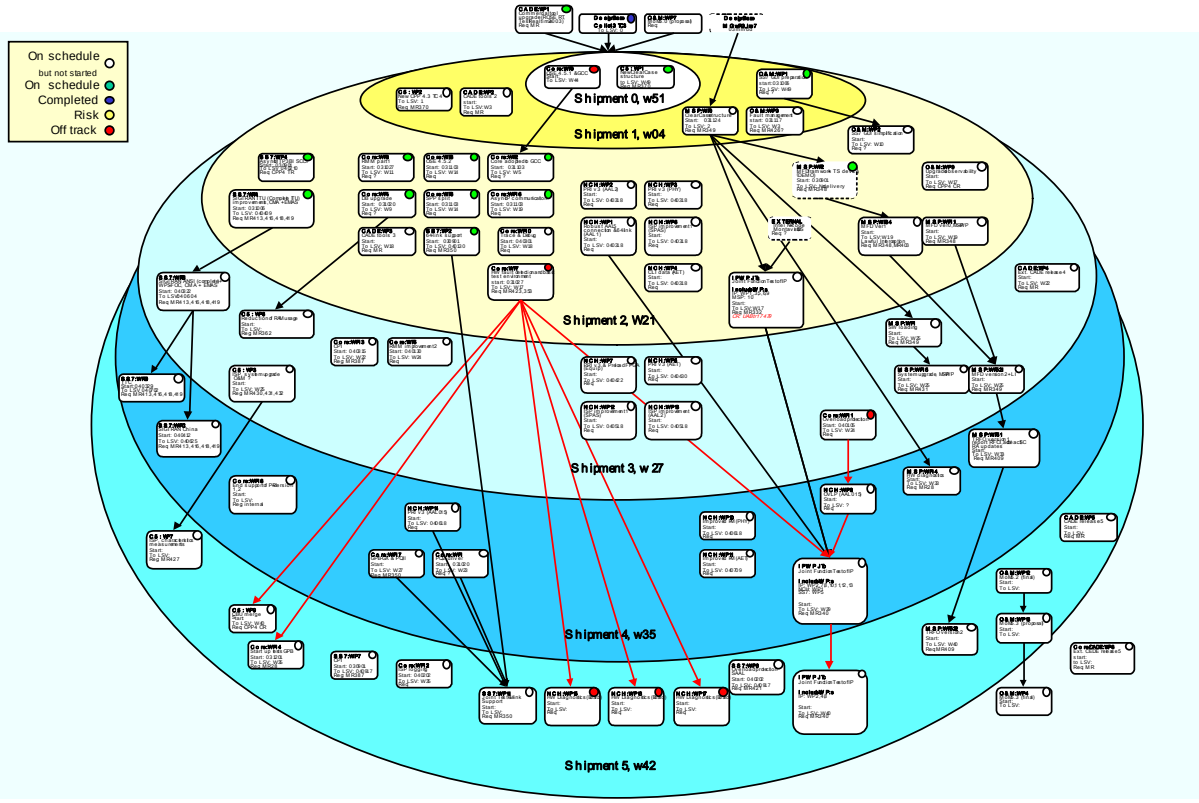

Figure 3: An example of the Δ anatomy

The dependencies between Δ:s constrain the order in which changes can be done, and determines the possible degree of parallelism. There are two types of dependencies that should be considered and managed:

- *"Design Dependency"* (ΔA should be delivered before *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"* (ΔA must be delivered before *verification* of ΔB can start). This dependency indicates that design of ΔB can start before ΔA is ready, but verification of ΔB can only be done after the completion and integration of ΔA.

The use of the Δ anatomy is based on several important principles:

- The anatomy should be easy to read and comprehend for all stakeholders involved. The intention is that the anatomy should be a common illustration of the tasks at hand. Thus, it becomes a means by which various decisions can be taken and evaluated.
- Developing the anatomy is a joint and continuously ongoing effort; it's not a one time effort done by a few specialists. Although the anatomy is an efficient way to communicate the work at hand, the process of creating and maintaining the anatomy is far more important and valuable than the resulting anatomy chart.
- The anatomy is the prime instrument for planning and monitoring the project. As such, there is a unique anatomy for each project geared precisely to the task at hand. During a typical project the anatomy will be revised at least once a week, reflecting the dynamic nature – new and changing needs and priorities – of system development.
- There should be only one anatomy for each system development project. If it turns out

that there is a need for several interconnected anatomies, the level of detail is wrong. The reason for maintaining this principle is again instrumental; there should be only one common overall view of the work to be done, and this view should be sufficient for all stakeholders involved.

Planning

The purpose of the planning activity is to decide what should be developed based on customer needs, how these needs should be transformed into suitable system changes (Δ:s), and in what order these changes are to be developed and integrated into the system. The planning is an ongoing process that results in regular releases at predetermined intervals. Thus, a "rhythm" is introduced in the development, something which appears to be advantageous for implementing large-scale systems (e.g. Lui and Chan 2008).

In the planning activity, three major, interrelated tasks are performed by system management, systems engineering and project management respectively (see Figure 4):

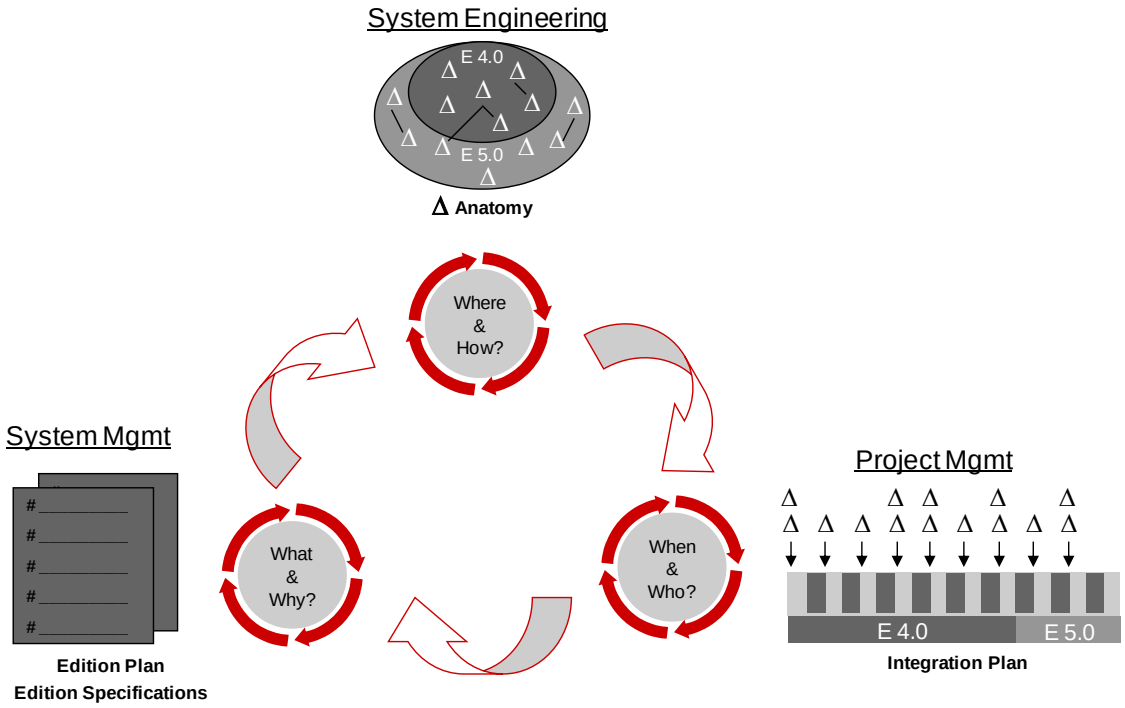

Figure 4: The anatomy-based planning

System management - What and Why? The purpose of this task is to specify the content of system editions – official system versions delivered to customers – in terms of needed functions and qualities, and maintain a system edition plan.

System engineering - Where and How? The system engineering activity transforms the edition content into small verifiable system changes (Δ) and clarifies their dependencies in a Δ anatomy. This includes specifying each Δ in terms of affected components and interfaces, and estimating its size.

Project management - When and who? The purpose of this task is to determine a suitable Δ integration order based on their dependencies and available development recourses, add these Δ:s into an integration plan, and kick off their realization by assigning Δ:s to development teams.

Realization

The realization activity can be regarded from two contexts: the *coordination context*, where the coordination of Δ deliveries are in focus, and the *development context*, where the development of a certain Δ by a Δ team is in focus (see Figure 5).

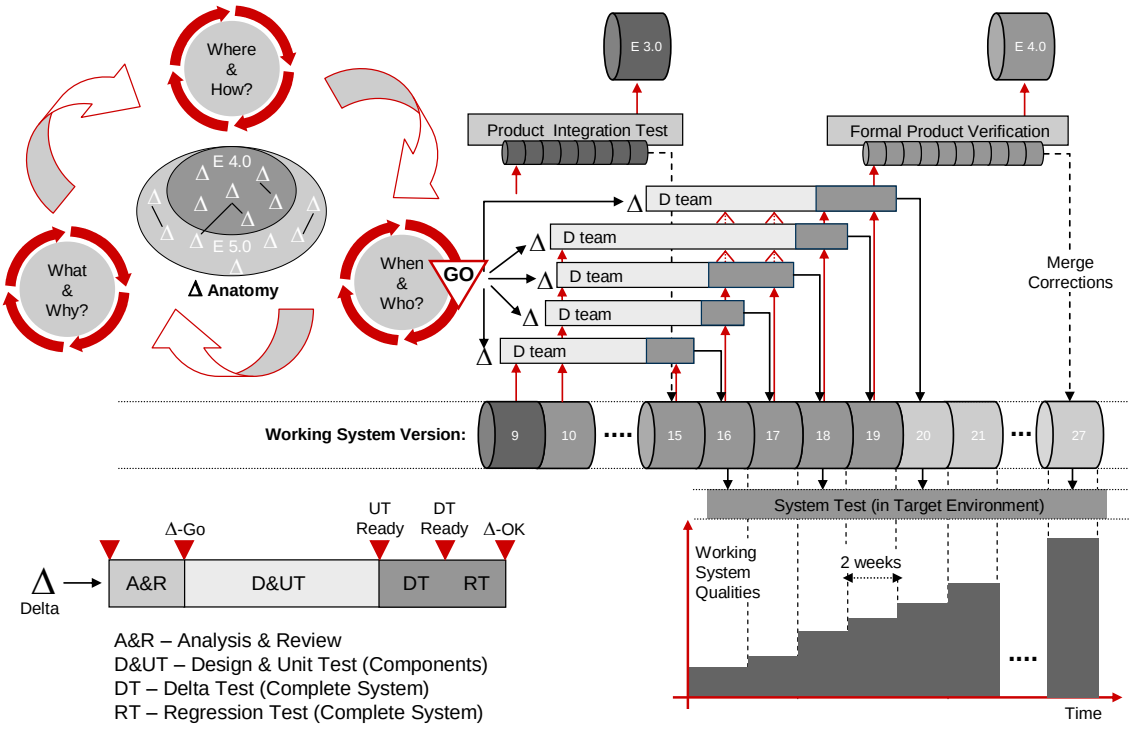

Figure 5: The realization activity

The coordination context. From the planning activity, the overall work has been divided into verifiable system changes, Δ:s, and an integration order has been planned. Each new system version resulting from integrating a Δ is called a Working System Version (WSV).

Figure 5 shows how new working versions of a system are produced frequently, (typically every second week). Each WSV is developed by a cross-functional team that has the end-to-end responsibility for its realization. The typical size of a development team is five to ten persons, and the development time may vary between a few weeks up to several months.

The task for a development team is to design, implement and verify a specified Δ. As can be seen in Figure 5, development teams must work in parallel in order to carry out the frequent (bi-weekly) integration pace. To avoid the situation of having several teams working with the same components in parallel, dependencies between deltas must be clearly understood – the Δ anatomy – and the order of realization carefully planned – the integration plan.

Before a development team is allowed to deliver the realization of a Δ, they must make sure that their new and modified components work in a complete system build based on the latest WSV. That is, development teams don't deliver components or sub-systems but always complete system versions that have been demonstrated to work as expected in a target like environment. This means that once a working system version is sent to System Test or Formal Product Verification we don't expect to find any trivial software faults. Note that Formal Product Verification *not* is done for each working system version, but only for system versions representing the implementation of a complete edition (for example WSV 19 in Figure 5).

The Δ team context. The procedure to be followed by each Δ team is outlined in Figure 5, the lower left part.

- *Assignment*: The Δ team leader has been asked by the project manager to start the realization of the Δ.
- Δ-*Go*: The Δ team has reviewed and analyzed the information specifying the Δ, identified the system components and documents that will be affected by the change, specified how the change will be verified (test cases), and agreed on a delivery date.
- *UT Ready*: The unit test is ready and all the changed and new components have been tested as agreed upon at the Δ-Go.
- Δ*T Ready*: The Δ test is ready and the Δ has been demonstrated to work in a system test environment without any major remarks.
- Δ-*Ok*: All quality assurance activities agreed upon at the Δ-Go, including Δ- and regression-test, have been carried out without major remarks, and the Δ team can deliver a new WSV. Regression testing is done to assure that previously working capabilities are still working.

A precondition for the Δ team is that the team really has an end-to-end responsibility (from "needed change" to "new system version"). This requires that boundaries are clearly stated in terms of criteria that must be fulfilled: 1) before an assignment can start, and 2) before a delivery can be made. Teams must also have the possibility to make a true commitment, that is, time for analysis, planning and negotiation. Moreover, teams are allowed to decide how to do the work, and everything should be done to provide them with the environment and support they ask for. Communication within and among teams ("team rooms", "daily stand-up meetings", "coordination meetings" etc.) are encouraged and facilitated.

The Δ teams should work according to agile methods. The realization of a Δ can, for example, be seen as one or several SCRUM sprints. The coordination context would then show a number of parallel, ongoing sprints. The teams are expected to use all kinds of agile methods such as refactoring, test-driven development, pair-programming, daily build, user stories, and so on. IDD does not prescribe any specific methodology or tools that must be used by the teams. This is up to the teams themselves to decide. Thus, the IDD can be seen as an approach where rigorous planning is combined with agile implementation.

Variants

The original IDD approach can be conveniently summarized as in Figure 6, where the following principles are adhered to: 1) "the work is divided into verifiable system changes", 2) "teams have an end-to-end responsibility", 3) "teams do verification before integration", 4) "product verification (i.e. a formal verification before a new edition is released), is done in parallel with development". There are, however, a number of variants of this approach that can be exploited during certain circumstances.

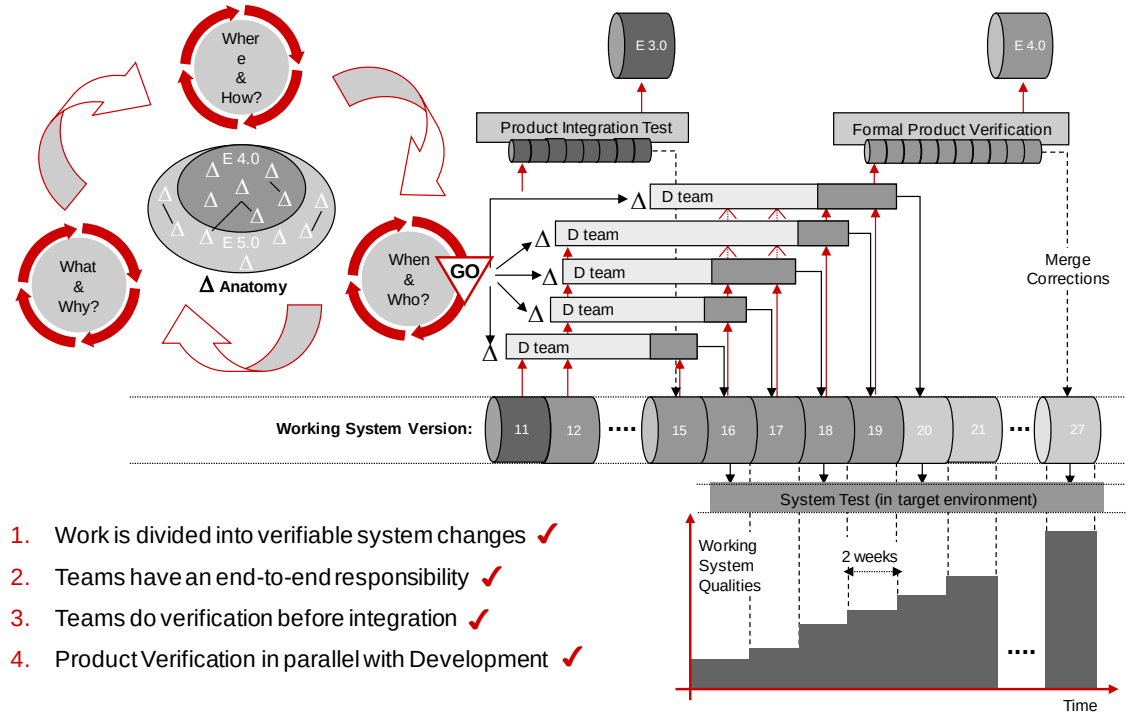

Figure 6: The original approach

Variant 1 - Sub-system teams

In the original approach, the teams are expected to be general, long lasting cross-functional teams that can take on the realization of any Δ, regardless which subsystems or components the Δ will affect. This requires that the ownership of the code and its design is "everyone's responsibility", that is, each team has the right (and ability) to do all the changes needed in order to get the realization of their current Δ done.

However, for large complex and technical systems, clear individual ownership and responsibility for each and every component is considered crucial in order to preserve the components' original architecture and "design idea" over time. Thus, for some systems (for example, telecom systems and avionics systems) it's more common to form teams and responsibility round components and subsystems, instead of having general development teams that are expected to do work everywhere in the system (see Figure 7).

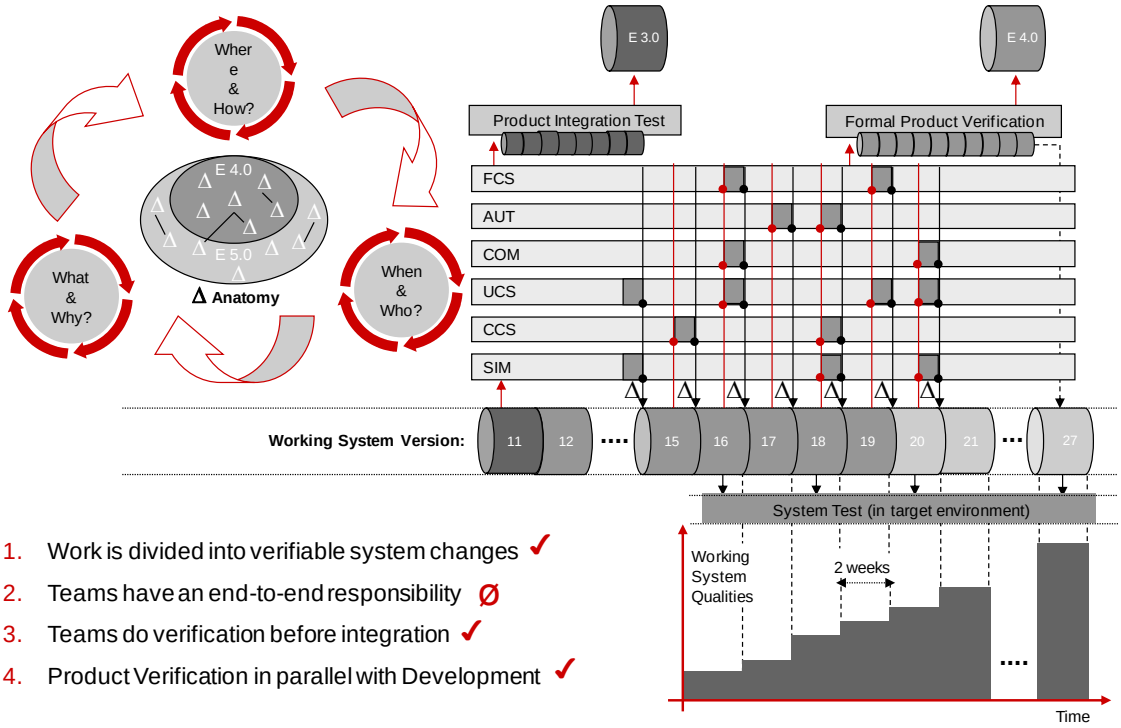

Figure 7: Forming subsystem teams

In this variant, the second principle "teams have an end-to-end responsibility" is not fully applied. Instead the quality and integrity of each sub-system are more emphasized. As a consequence, each Δ delivery must be coordinated between impacted subsystem teams. For example, the Δ delivered to WSV 21 in Figure 7, impacts subsystem teams SIM, UCS, and COM. Before these three teams are allowed to deliver their modified subsystems, they must do a common test to show that their subsystems work together, that is, to show that the new system version works as expected, and that the complete Δ is properly implemented.

Variant 2 - "Daily build"

Applying the third principle "verification before integration" means that each team must work in isolation from the official WSV, that is, work on an own development branch and not directly in the WSV branch. The main advantages of this are: 1) It's possible to always have full control of the content and quality of the WSV, 2) it's possible to halt the realization of a Δ without any extra work of "removing" things from the WSV, and 3) teams can *choose* when they want to rebase their design base with the latest official design base (WSV) instead of being forced to always use the latest "checked in" code for testing activities.

However, having teams working on their own development branches requires a modular system architecture, with clear interfaces, and subsystems and components implementing well defined parts of the overall services provided by the system. If, from a total system perspective, every valuable and verifiable change (Δ) typically requires changes to a majority of the components and subsystems, 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; thus avoiding the risk of spending most of the development time merging component versions developed in parallel.

This variant of stepwise development of large systems can be viewed as the well known "daily build approach" applied on the complete system level; perhaps not daily but frequent (see Figure 8).

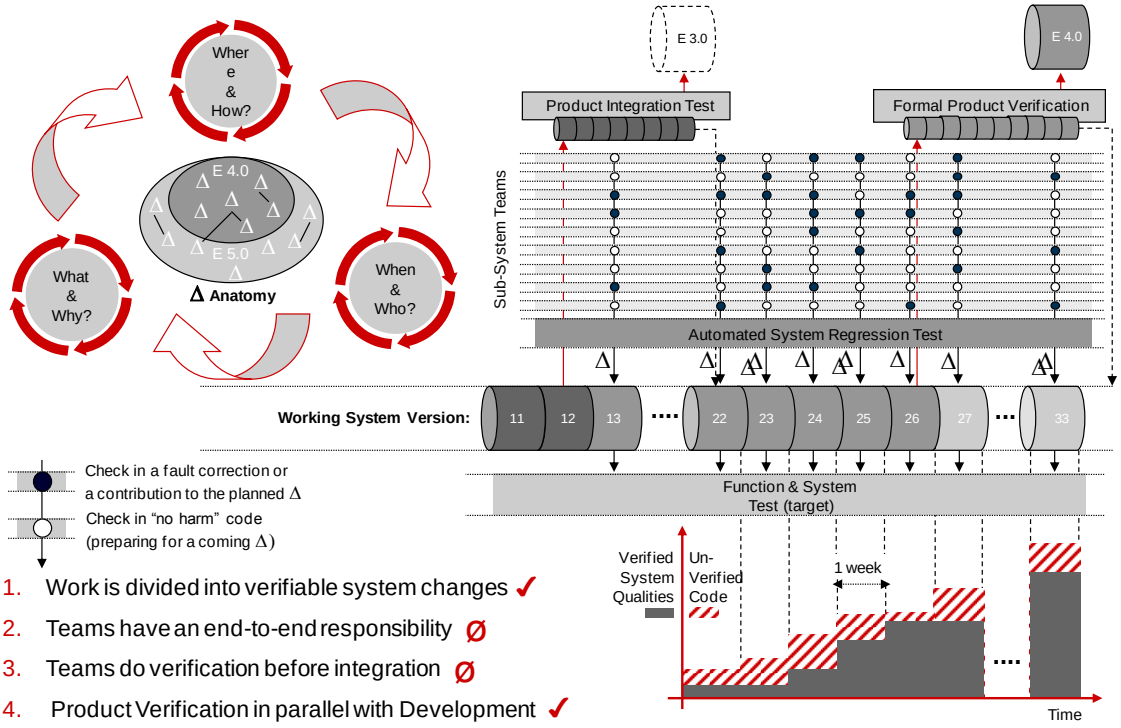

Figure 8: "Daily build"

In the "daily build" approach the planning process is similar to the original approach. The development scope is divided into verifiable system changes (Δ), and the integration plan shows in what order and when the Δ:s are expected to be implemented and ready for function test.

The WSV is a weekly (or biweekly) created baseline in the one and only code branch. Every night a complete system build is done and an automated regression test is run on the latest checked in version of all components. The quality and coverage of the regression test determines the quality level of the WSV. In the best of worlds, the automated regression test will check that basic functions, which worked in the latest edition or *released* WSV (for example WSV no.12 in the picture above), are still working. So, when establishing a WSV baseline, we know that basic functions still work. However, it remains to check whether the newly integrated Δ works. In the "daily build" approach "function test" is done *after* Δ integration, not *before* as in the previous two approaches.

Moreover, in the "daily build" approach, the second principle "teams have an end-to-end responsibility" is obviously not applied. Development teams are delivering components without knowing if the components are good enough to result in a new working system version with planned function and quality. In this approach, delivering working system versions is typically the job for a test- or integration-team on system level.

Discussion and Conclusions

Agile methods emerged as a well justified reaction to the "heavy-weight" 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 increasing scale of projects, concerns with "agile-only" methods begin to show. IDD might be seen as yet another step in the evolution of methods, combining rigorous 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 (Boehm and Turner 2003); the answer of which is – both. With increasing scale of system development, an element of stability in agile methods is indispensible.

A critical element in the IDD approach is the Δ anatomy. The anatomy construct has remained in one form or another all through the years at Ericsson since it was first conceived almost two decades ago. The main reason for this is undoubtedly that it visualizes the most important things when developing complex systems: how capabilities in the system depend on each other. If a capability in the chain of dependencies is not present, the system will not work as intended.

Moreover, the anatomy is consciously kept as simple as possible in order for it to be easily understood by all stakeholders in the project. The importance of shared or common understanding cannot be overestimated. If there is no common picture of the system to be developed, the risk of failure is imminent. Because of its focus on communication rather than technical details, the anatomy is one way of approaching the social dimension of ULS systems.

As with any approach, IDD of course has its drawbacks. The approach is hard to implement, and requires a long-lasting, committed effort to "stick" in the organization. This has been evident in efforts to transfer the IDD to other organizations than Ericsson. The basic principles of the anatomy, planning procedure and execution of Δ:s, are easily comprehended. Putting these to work is an entirely different matter.

Concluding the paper, the main point about IDD can be expressed as follows. If there are many agile teams or "sprints" going on simultaneously, and all teams deliver parts of the same system, then it's necessary to spend time on how to coordinate these teams. This effort concerns primarily dividing the development scope into suitable changes or additions (Δ:s), and deciding their 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. Hence, combining agile development practices with plan-driven methods is not only possible; it's probably necessary in order to improve large scale development performance.

References

Adler, N. 1999. *Managing Complex Product Development – Three approaches*. EFI, Stockholm School of Economics. ISBN: 91-7258-524-2

Anderstedt, J., Anderstedt, U., Karlsson, M., and Klasson, M. 2002. *Projekt och helhet – att leda projekt i praktiken*. Författares Bokmaskin (in Swedish), Stockholm, ISBN: 91-7910-380-4.

Boehm, B. W., and Turner, R. 2003. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.

Fraser, S., Boehm, B., Järkvik, J., Lundh, E., and Vilkki, K. 2006. How Do Agile/XP Development Methods Affect Companies? In *XP 2006 LNCS 4044*, ed. P. Abrahamsson, M. Marchesi, and G. Succi, 225–228. Springer-Verlag: Berlin Heidelberg.

Gilb, T. 1989. *Principles of Software Engineering Management*. Addison-Wesley Longman.

Jönsson, P. 2006. The Anatomy-An Instrument for Managing Software Evolution and Evolvability. In *Second International IEEE Workshop on Software Evolvability (SE'06)*, 31-37. Philadelphia, Pennsylvania, USA. September 24, 2006.

Karlsson, E. A. 2002. Incremental Development- Terminology and Guidelines. In *Handbook of Software Engineering and Knowledge Engineering, Volume* , 381–401. World Scientific.

Larman, C., and Vodde, B. 2008. *Scaling lean & agile development: thinking and organizational tools for large- scale Scrum*. Upper Saddle River, NJ: Addison-Wesley.

Lui, K., and Chan, C. C. 2008. *Software Development Rhythms: Harmonizing Agile Practices for Synergy*. Wiley

McConnell, S. 2004. *Code Complete*. 2nd edition. Microsoft Press. ISBN 1-55615-484-4.

Meso, P., and Jain, R. 2006. Agile Software Development: Adaptive Systems Principles and Best Practices. *Information Systems Management*, 23(3), 19 – 30.

Royce, W. 1970. Managing the Development of Large Software Systems. *In Proceedings of IEEE WESCON 26* (August), 1–9, Retrieved March 9, 2010, from http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

SEI Carnegie Mellon. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. ISBN: 0-9786956-0-7. Retrieved Nov 30, 2009 from http://www.sei.cmu.edu/library/abstracts/books/0978695607.cfm?WT.DCSext.abstractsource=RelatedLinks

Standish Group International. 2009. *CHAOS Summary 2009*. The 2009 update to the CHAOS report. Retrieved Nov 30, 2009, from http://www.standishgroup.com

Taxén, L., and Lilliesköld, J. 2005. Manifesting Shared Affordances in System Development – the System Anatomy. In ALOIS*2005, *The 3rd International Conference on Action in Language, Organisations and Information Systems*, 15–16 March 2005, Limerick, Ireland, 28-47. Retrieved Nov 30, 2009, from http://www.alois2005.ul.ie/

Lars Taxén, Associated Professor– Has more than 30 years of experience from the telecom industry, where he has held several positions related to processes and information systems. His thesis concerns the coordination of large, globally distributed development projects with focus on 'soft' issues like sense-making. He has written a book, several book chapters, and published in various conference proceedings and journals. He is now active as a researcher and consultant (homepage: www.neana.se).

Ulrik Pettersson has worked 20 years in the systems and software engineering business taking on various roles such as system designer and project manager. For ten years Ulrik worked at Ericsson with the forming and establishment of Ericsson's system development strategies. Ulrik is now managing a unit at Saab Aerosystems responsible for strategies, methods and tools used to develop safety critical avionic systems.