Architecting for asynchronous developments and life cycles

White Paper Resulting from Architecture Forum Meeting

November 15-16, 2017, Sioux Eindhoven, The Netherlands

Edited by:

Teun Hendriks, ESI

Gerrit Muller, USN-NISE and ESI

Eirik Hole, Stevens Institute of Technology

Input was provided by the following participants in the Architecture Forum:

Name	Organization
Maarten Bisschoff	Thermo Fischer Scientific
Maarten Bonnema	USN-NISE
Goran Hansson	Mycronic
Teun Hendriks	ESI
Eirik Hole	Stevens Institute
Kees Kooijman	Sioux Embedded Systems
Hans Kuppens	Sioux Embedded Systems
Bjørn Victor Larsen	Kongsberg Defense
Wouter Leibbrandt	ESI
Alexander Lepple	Daimler AG
Roland Mathijssen	ESI

Name	Organization
Jamie McCormack	Thermo Fischer Scientific
Gerrit Muller	USN-NISE and ESI
Andre Nieuwland	Philips
Bob Reinke	Roche Diabetes Care
Martin Simons	Daimler AG
Lauri Ståhle	Patria
Marnix Tas	Sioux Embedded Systems
Hugo van Leeuwen	Thermo Fischer Scientific
Bruno van Wijngaarden	Vanderlande
Martin Verheijen	Thermo Fischer Scientific
Paul Zenden	Sioux Embedded Systems

Published, November 2018

1 Introduction

Several trends in systems development increase the diversity of development and change rates within the system and in its context. The ever-increasing integration and interoperability, and the ever-increasing rate of technology developments are examples of such trends. For example, many "physical" systems are nowadays connected to the cloud and to apps running on smart phones and tablets. At the same time, they are part of Systems of Systems. The effect is an increase of diversity of heartbeats and a decrease of the control; many systems and components may change outside the architect's scope of control.

The members of the architecting forum discussed this topic, using the following questions:

- How much context do architects need to cover?
- How can architectures facilitate various heartbeats in the system life cycle?
- What architecture patterns cope with variation in heartbeats?
- How to cope with variation in heartbeat of disciplines?
- What is the relation to continuous delivery?

2 Time scales in operation of systems, and change rates of development

Over the past decade, system time scales have decreased significantly both in operation as development of systems. Yet at the same time, diversity in time scales has widened, both inter-system, and inter-organization.

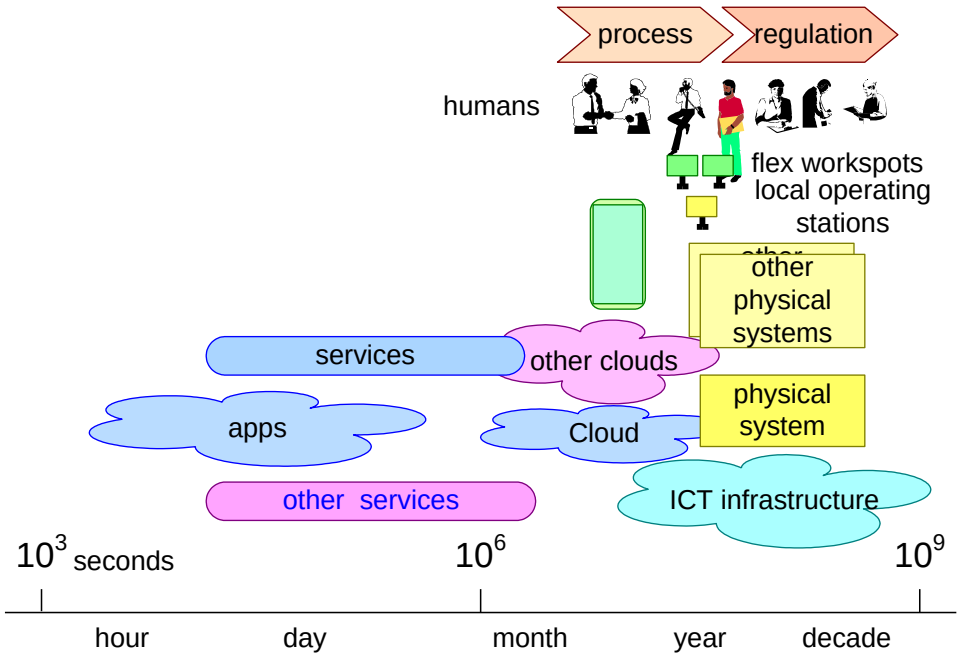

Figure 1: Typical variation in system development lifecycles on a time scale of seconds (Muller, 2018)

Figure 1 shows an example of time scale variations in system development lifecycle heartbeats (taken from (Muller, 2018). To relate these widening time scale variations to challenges on architecting and architects, an inventory was taken on current times scales as practiced both in operation of systems as change rates in developments and lifecycles.

Table 1: Time scales in System Operation and System Development (items ordered in increasing time scales)

Time scale	System Operation	System Development
Nano-second range	• Micro-electronics operation speed • Acquisition time per pixel • Move to next pixel time	
Micro-second range	• Micro scan • Electron-beam repositioning • Gateway routing of messages • DB access timing • Servo control loop rates	
Milli-second range	• Wafer die illumination • Camera frame rates • Typical event rates in systems • Sensor pulse rate for on-body medical measurements	
Second range	• Wafer handling • Write stroke • Change rate for adaptive medical treatment delivery • HW data logging	
Kilo-second range (¼ hour to ~12 days)	• Experiment time for 3D reconstruction • Analytical techniques for Life Sciences • Change rate in management systems • Wafer mask production • (Re-) Calibration of precision equipment	• Rate of incoming errors/field reports • Rate of incoming change requests / feature requests • SW patch release
Mega-second range (~12 days to ~30 years)	• Precision equipment maintenance rate • System operation interruption • Failure of cloud infrastructure • SW update • HW update/refurbish • Major service activity	• New SW release / new feature release • New cloud release / new APP release • New system-of-systems feature • HW module development • Medical device development and approval by authorities • Major revision of product platform
Giga second range (30+ years)	• Precision equipment end-of-life	

This Table 1 lists time scales from nanosecond (10^{-9} second) to gigasecond (10^{+9} second, i.e. 30+ years) for both system operation and system development. Table 1 shows that times for the most elementary *system operations* have moved into the nanosecond range, while the most complex operations remain in the second to kilo-second range, despite their increasing

sophistication. Conversely, service & maintenance time scales (system downtimes for repair, servicing, and equipment calibration) are to be pushed further out in the mega-second range. This table also illustrates for *System Development* the increasing variation between HW and SW development. Weekly SW patch releases have become commonplace, whereas (specialized) HW development of critical modules may take many months to years. This also requires a critical view for which purpose to develop specialized HW in-house, and where to use, i.e. rely on commodity HW.

Business trends such as the increasing connectivity of systems to the internet are a further source of variations in system development time scales. New features and SW updates, new cloud or APP versions may follow each other in a matter of months, whereas new development of regulated devices (e.g. medical devices) may take years. For professional equipment a major overhaul of a product platform is feasible only once every 10-15 years. Nonetheless, individual HW modules may undergo several revisions within this period.

3 A case of asynchronous development: decoupling customer project engineering from product (platform) development in warehousing

To start off the discussions, a forum member presented a case study on warehouse management systems. The warehousing business has seen tremendous growth, for a large part due to growth in E-commerce business. Online shopping is now ubiquitous. Shoppers expect the on-line ordering and delivery process to keep pace with their busy lives. Thus, same-day deliveries, or even within a few hours are an important driver in this business. Figure 2 shows the typical goods workflow in warehouses. On the receiving end, new supplies come in, and replenish stocks in storage. Customer orders trigger picking of purchased items, which are next packaged and forwarded to shipping.

To meet the rapid growth with also rapid change in customer demand, these warehouses need increased flexibility and speed in order handling. Also new installations need to be realized on short notice. These drivers have caused a shift from ‘pure’ project engineering (made-to-order warehouse systems) to asynchronous product platform development with upfront product platform definition and development of building blocks. In this shift, project execution (building, configuring, and installing a new warehouse management system for a specific customer) has been decoupled from product platform development.

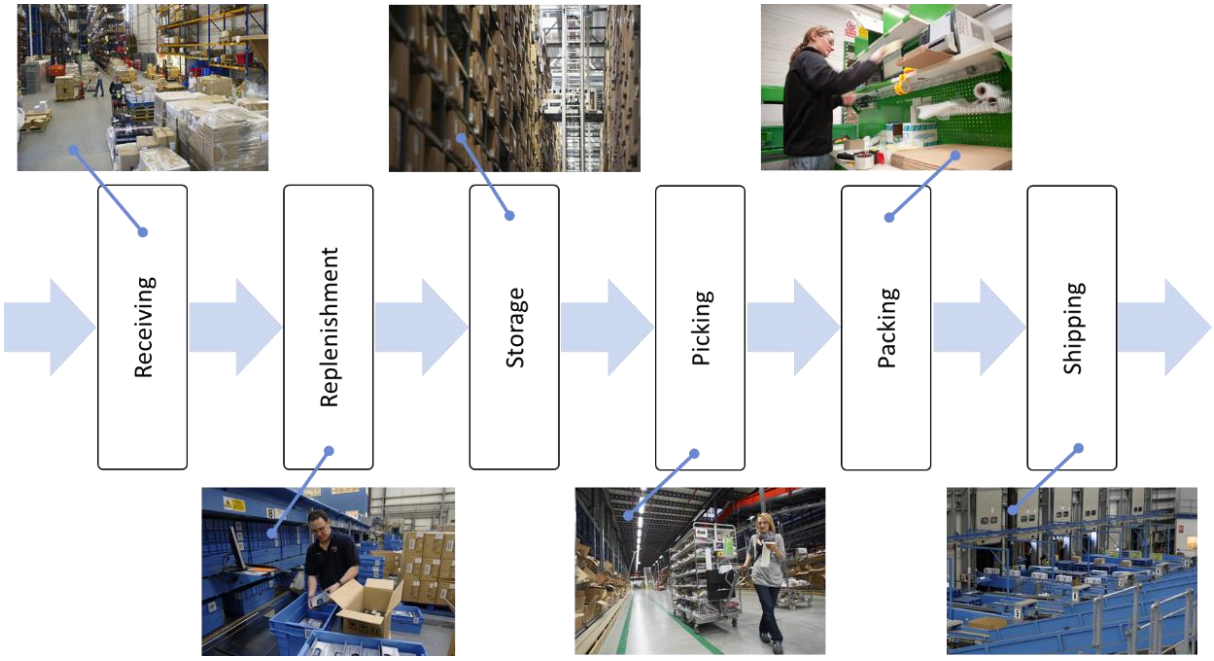

Figure 2: Typical process flow in a warehouse management systems

For the product platform development to be able to operate asynchronously from system projects, non-recurring project engineering costs needed to be limited and customer-specific solutions restrained. The approach taken is to go for a reference architecture with judiciously chosen building blocks with minimal interfaces, and minimal overall system requirements. Most requirements are transferred to module level. The general strategy to simplify complexity is by careful definition of building blocks to isolate sub-systems. Domain-driven design (Evans, 2004) is used in creating re-usable building blocks (see also Figure 3). Customer specific requirements are preferably solved in specific add-on modules or handled in SW. Furthermore, architects may trade off some penalty in performance in order to keep close to the reference architecture, and avoid project specific development., Motivators and enablers are needed to ensure success of a platform approach: specifically, tools and presentations are needed to help sales and product development understand and adopt the platform.

The challenge for architects in this transition is to change from a role as experienced system designer to a designer and guardian of the product platform, managing expectations of many stakeholders. The definition of the reference architecture is crucial; trade-offs in this architecture determine the scope that it covers. If not well thought out, this could cause business trouble and thwart asynchronous product development towards system projects.

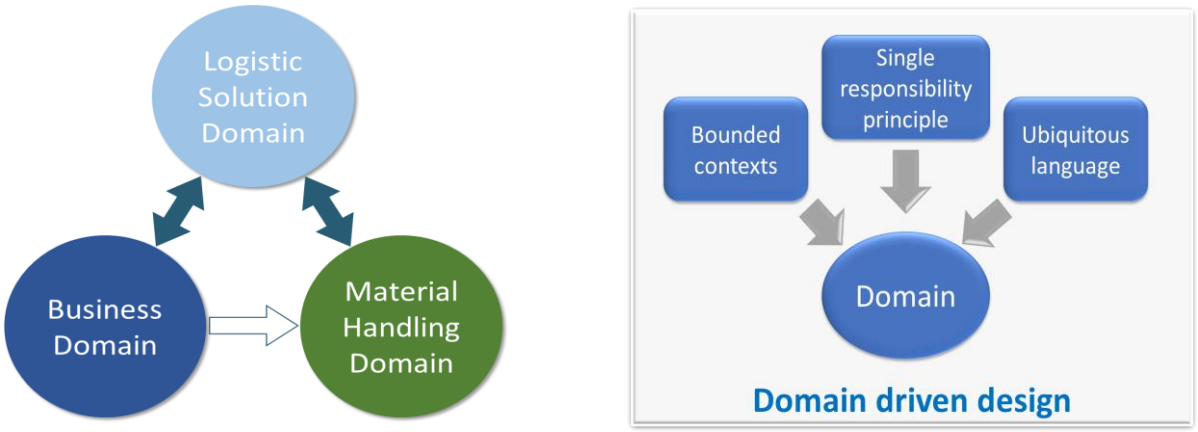

Figure 3: The three key warehousing domains (left) considered in the domain-driven design approach (right)

In the discussion consensus arose that variations in life cycles are inevitable in complex systems, yet it brought about several additional questions and issues for architecting:

- Are "Asynchronous" and "Different life cycle periods" the same or not?
- Customers are generally not prepared to pay for "under-the-hood" engineering. How to deal with necessary platform infrastructure?
- How to deal with crosscutting concerns such as security and reliability, while still respecting bounded contexts, and modularity in general?
- How to cope with market discovery (some new feature induces a yet unknown need). Such market discovery is probably not predictable.
- Must asynchrony be treated as a problem, or can it be taken advantage of (in the way that Google and Netflix do)? Clearly, this is more difficult with physical systems than purely digital systems, but analogs could be drawn.

4 Causes for asynchronous development and development lifecycle variations

In asynchronous development, customer project/systems development is decoupled from product platform and component development. Asynchronous development in systems usually appears because of significant differences in cost, effort and lead-time of functionality and/or constituting system components for innovations or urgency of identified business opportunities.

The "Diamond of Innovation" model (Shenhar 2005) provides for a categorization of such factors (see Figure 4) and was developed as an aid to select the right form of project leadership for development of innovations. The model distinguishes the following four categories:

1. “Novelty” – How intensely new are crucial aspects of the innovation and project?
2. “Technology” – Where is the innovation positioned on the scale from low-tech to super high-tech?
3. “Complexity” – How complicated are the product, the process and the innovation?
4. “Pace” – How urgent is the work? Is the timing “normal, fast, time-critical or blitz”?

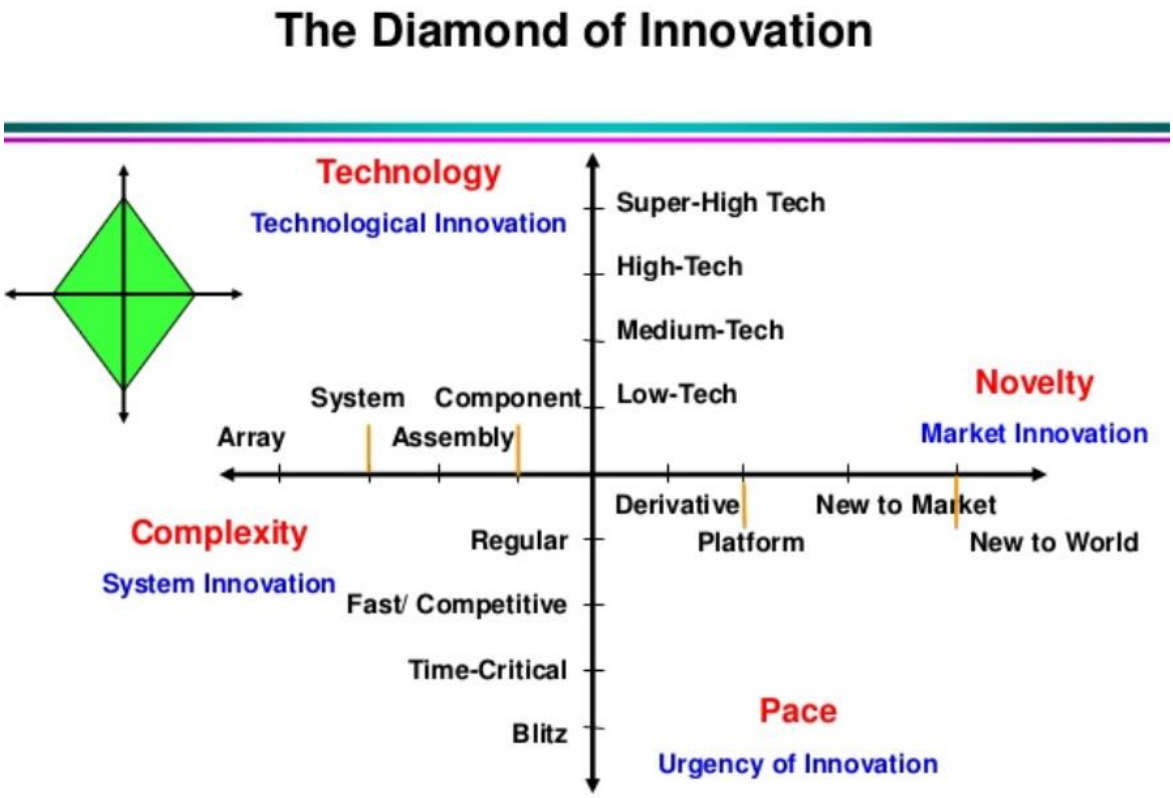

Figure 4:The diamond of innovation model: Four dimensions of project uncertainty that have to be managed.

This model can be applied to position innovation development within the organization (a second use is to position the organization with respect to the outside world, and other organizations). Variation in these categories, or in their combination, typically gives rise to variations in lifecycles and development. In the following, we classify drivers for asynchronous development using these categories are elaborated.

4.1 Level of novelty

The ‘level of novelty’ addresses the level of market uncertainty and it impacts the effort and time it takes to define the product’s requirements clearly. The level of novelty may vary from a derivative product with low-risk revisions and feature enhancements to “breakthrough” products not having a defined market and customer base yet:

- Derivative products incorporating incremental updates to a successful product line in general can be well-planned and typically run on a fixed cadence. The cadence of component change may vary given the varying cost of change of SW, versus that for dedicated HW, or commodity HW.
- Platform updates, i.e. an overhaul of a product concept and infrastructure, take place only every 10-15 years for complex systems. These are complex undertakings as they still must address the existing customer base and needs, as address new market trends and counter competitive offerings.
- New products or features with rapidly changing customer requirements, unknown market requirements, or varying market readiness levels typically need shorter iterations and rapid prototyping towards a Minimum Viable Product.

With complex systems, the level of novelty may also vary from subsystem to subsystem. For example in an Electron Microscope, the core electron beam functionality may be stable, whereas image processing and analytics capabilities may undergo more rapid innovation.

4.2 Level of technology newness

The 'level of technology' addresses how much new or leading-edge technology is used. It represents the level of technological uncertainty or technological update rates.

- Systems or components operating on the leading edge of technology e.g. physics are more difficult to develop than low-tech systems.
- The stability of the technology, its rate of evolution, may drive also system or component updates.
- On the other side of the spectrum, technology obsolescence drives change too.

Low-tech innovations have almost no technological risk, but they require maximum efficiency to gain returns. As the level of technical complexity increases, so does the risk of failure and the likelihood that a strive for maximum efficiency in development will backfire.

4.3 Level of complexity

The 'level of complexity' addresses how complex the system and its development are. Complexity impacts the degree of formality and coordination needed to manage an innovation step effectively. Level of complexity may vary based on the following factors:

- Type of development: SW versus HW, commodity HW versus dedicated HW.

- Regulatory regimes: safety-critical devices and systems have strict verification and validation processes and lengthy approval cycles.
- System performance requirements versus vs technology stack.
- System complexity in general: large complex systems versus small systems.
- System break-down in modules, and amount, and variety of module suppliers.

4.4 Required pace of innovation

The pace of innovation drives the urgency or rhythm to complete innovation or updates. It impacts the time management and autonomy of the project development team.

Factors driving the pace of innovation or updates may be the following:

- Sales cadence (trade shows, Model Years, Xmas sales) may set the pace of updates.
- Asynchronous appearances of issues (planned versus emerging):
    - Blocking “in-field” problems (e.g. safety risk) have high priority.
    - Field feedback leads to new features/gaps in current features.
- Varying customer demands come in at different times.
- Time to market aspects:
    - (Faster) changing market requirements.
    - Response to competitive offerings.

4.5 Consequences and examples

Variation in above aspects are causes for asynchronous development and variation in development cycles. Examples of systems with variation in development life cycles are numerous. Hardware, firmware, application software, user interface, operating system, cloud based, different sub-systems like frames, actuators, electronic boards and components, suppliers and personnel, all have their own life cycles.

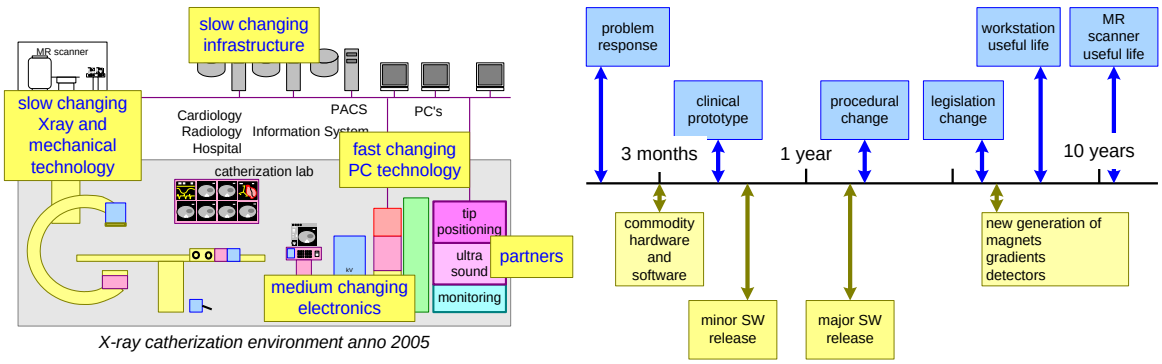

Figure 5: Example of technology change rates & life cycle variations for medical equipment (Muller, 2005-2018)

In e.g. medical equipment development, SW patch releases may be issued weekly; SW updates with new features released bi-annually, and HW updates only once every few years; SW updates also are commonly offered for systems in the field (see Figure 5). The trade-off with managing such asynchronous development is balancing new features versus the stability of the system operating in the field.

In the automotive industry, vehicle maintenance in the garage now not only involves an oil change, but also a SW change. Companies such as Tesla are even able to update over-the-air most if not all SW inside its vehicles, even those with deep impact on the vehicle's driving behavior, e.g. changing SW to significantly reducing stopping distances on the Model 3 in response to Consumer Reports complaints (The Verge, 2018).

5 Architecting for asynchronous development

Development approaches such as Product Line Engineering (Van der Linden et. al., 2007) and Agile Development (e.g. Scaled Agile Framework) address process needs for asynchronous development and target more rapid response to market needs. Yet the architect and architecture need to make such asynchronous development possible. Leffingwell (Leffingwell, 2007) calls this the required context of an *intentional architecture*. Large systems, consisting of many subsystems, require an *intentional architecture:* an explicitly designed, component-based architecture, where each component can be developed as independently as possible and which conforms to a set of purposeful, systematically defined interfaces. Agile component teams must be able to fit their components into this architecture, and this architecture should be aligned with the development teams's core competencies, physical locations, and distribution.

Asynchronous development requires an architect to have a pro-active attitude and to plan further than just the next release or model. Considering the longer time horizon, architects will also need to align extra with their stakeholders, who may have different priorities and timescales of their own. Hence, the key challenges architects phase with asynchronous development and life cycles are the following:

- To understand the future, both market and technology wise.
- To plan ahead: align development steps and feature roll-out with stakeholders.
- To manage system partitioning and interfacing to support effective development increments and asynchronous lifecycles.
- To fit in new developments and unforeseen market needs as they arise.

5.1 Understand the future

Architecting for asynchronous development and lifecycles means firstly to get a grip on what the future may hold in terms of market developments and evolving requirements, technology lifecycles, and secondly, to bring these, their relations, and their impact on the system together in a technology roadmap. Similarly, the long-term desired structure, patterns, and vision for the system can be documented in a reference architecture. Finally, early deployment of a Minimum Viable Product (MVP) can provide first market feedback rapidly to guide the further development process.

Market developments and evolving requirements. Architects need to understand market developments and evolving requirements. How are their systems incorporated in e.g. the production line of customers? What are new market requirements? When is the market ready for a new feature? To some extent, also the market will tend to discover its needs, based on product usage. Such market discovery is probably not predictable, and reason to support faster update rates and cycles where possible.

Technology evolution and lifecycles. Technologies as used in a system have their own lifecycles. Computing components typically evolve much faster than the large complex systems themselves. Component obsolescence is hence an issue to be understood and managed. Looming component obsolescence can be planned for with form-fit-function replacements. Component obsolescence may not only affect HW, also e.g. third-party SW may also be phased out by SW suppliers, and hence may become obsolete too.

Evolution rate of technology is also a factor to monitor in case of evolving technologies. E.g. with computing technology a shift from FPGA and ASICs to general purpose (multi-core) CPU has been made possible over the past 10 years. In automotive, battery technology is rapidly improving range of electric vehicles at affordable cost levels. Monitoring such technology trends is important to understand when they would cause a change in system concept, and to understand which technology suppliers to work with.

Roadmaps. Roadmaps provides a structured (and often graphical) means for exploring and communicating the relationships between evolving and developing markets, products and technologies over time. The result, the roadmap, is a tool to bring together the various elements of the future of a system in its market and/or society context. Roadmaps can take many forms, but the most general and flexible approach comprises a visual time-based, multi-layered chart (Phaal, Muller, 2009).

A key success factor for road mapping is ownership by specific individuals of both the roadmaps and the road mapping process. A technology roadmap typically represents a simplified, synthesized view of a complex system, and so the overall strategic roadmap is often owned and edited by an architect.

An earlier SAF meeting (Muller and Hole, 2015) covered the topic of road mapping.

Reference architectures. Reference architectures (Muller and Hole, 2007) capture the essence of existing architectures, and the vision of future needs and evolution. Having a reference architecture provides guidance to assist in planning the architecture for a product platform and evolutions thereof. Similar and complementary to a roadmap, it supports making of strategic decisions on innovation cycles and steps.

Reference architectures must also have some means to reason about performance within the platform, and the edges of design (which may be defined by the functional modules that are part of the platform). Clarity in scope is needed for feasibility of a platform approach.

An earlier SAF meeting (Muller and Hole, 2007) covered the topic of reference architectures.

Usage profile data and (early) field feedback. A typical way to go to the market fast and obtain early adopter feedback is to define and release a Minimum Viable Product (MVP). Subsequent upgrades (HW and SW) can be delivered in subsequent releases and extensions. Feedback from early adopters provides insights for new market needs and features.

A further key trend is to retrieve actual usage data from connected systems. Common practice in the digital world and APPs, this trend also extends to professional systems e.g. medical equipment and vehicles. To be able to do this efficiently, the necessary architectural infrastructure needed must be laid out in the first version of the product. Also, a back-office needs to be arranged. Care must be taken that the usage data collected is rich enough, and descriptive enough, to allow comparison of actual usage versus foreseen usage at time of development. These insights may drive further development.

5.2 Plan ahead

The obtained understanding of the (foreseeable) future should drive the making of a plan forward. This planning is typically coordinated effort of Strategic Product Marketing en Product Development, with a key role for the architect to arrive at a feature rollout plan, and development cycles. This planning must address market needs and technology developments, balance and align the various stakeholder needs and priorities, and align cycle speeds and heartbeats of various development cycles.

Stakeholder alignment and feature rollout planning. A key planning deliverable for enabling asynchronous development is a feature roll out plan. A feature rollout plan positions and prioritize market needs and release of new features over time. Stakeholder alignment on time to market may be required to achieve harmonization in the release schedules for development efficiency.

Development roadmap. The feature rollout plan is one key ingredient for a development roadmap. The second ingredient is the positioning of necessary system infrastructure development items and replacement planning for obsolete components. Incorporating planning for looming obsolescence requires impact analysis and definition of adopt/not adopt moments ahead of time, to secure budget and development effort in the case of "adopt" decision. On the other hand, HW preparations could be scheduled ahead of time if other needs would make a component upgrade opportune in a cycle.

The combined result should yield an aligned development roadmap with various modules and updates being released at different moments. This development roadmap forms a contract between various development groups and product marketing. The contract has meaning that if a group is in danger of not making the schedule, overtime may be observed to live up to the commitment. For precision equipment, such a development roadmap is typically made for 6 months increments over for three-year development look-ahead period (on system and major sub system level) and maintained/adjusted after every increment.

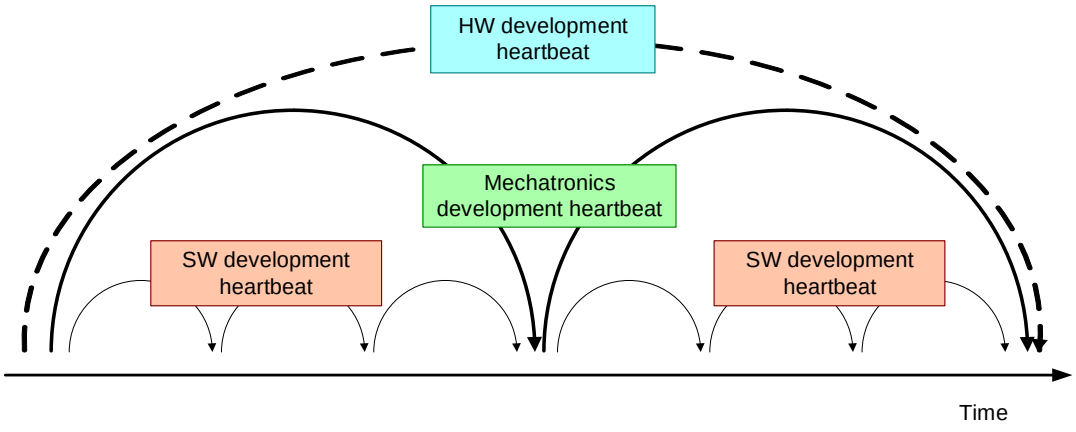

Figure 6: Alignment of e.g. discipline development heartbeats as integer multiple periods

Alignment of development cycles and heartbeat. SW development for incremental features typically can proceed much faster than development of dedicated HW components. The development roadmap must incorporate different update cycles and align these. Typically, development cycles are aligned on basis of integer multiple periods (see Figure 6) where:

- scope (goal-based alignment) drives the slower cycles,
- the slower cycles to drive the faster cycle content,
- continuous integration ensures that the baseline is kept stable.

Internal development cycles could proceed faster than external release cycles. A stable baseline can be maintained by continuous integration: multiple streams deliver into integration phase, also those that span multiple sprints. All streams jointly fix all integration issues that are found. Streams that span multiple sprints also deliver and integrate, and possibly their functionality is de-activated for the users. The shorter the internal cycles are, the less the differences to the (stable) baseline are, and so the less integration issues will be present. The stabilized integration result can be released to the field, but a release has release costs associated with it. It is a business decision how often to do this: it depends on the release costs where optimum between frequency and costs lies.

5.3 Manage system partitioning and interfaces for asynchronous development

To enable asynchronous development technically, a platform approach needs to be in place. Such a platform should comprise of set of (re-usable) functions/modules and an integration framework which together span the solution space. This poses a restriction in a way, but as the interfaces between the modules are known and stable, individual systems with predictable behavior could be generated in a short time-to-market, which is a tremendous benefit. The success of a platform approach and product line engineering hinges on a good support for the business drivers, a clear platform/reference architecture, supporting processes, and an aligned organization to achieve an effective product line engineering capability (van der Linden et. al., 2007).

System partitioning. Typical drivers for system partitioning are functionality, modularity, flexibility, and preparedness for future changes. Besides these, life cycle should also be a criterion for system partitioning. Slow-changing, expensive-to-change system parts should be separated from fast-changing, flexible system parts or functionality. A trade-off to be made is product cost efficiency and total cost of ownership over the life cycle. The origin or cause of different life cycles, i.e. a necessary change, may be diverse as explicated in section 4.

Principle 20.1: Life cycle is a criterion for system partitioning. Slow-changing, expensive-to-change system parts should be separated from fast-changing, flexible system parts or functionality.

The general strategy to simplify complexity is by good definition of building blocks isolating a sub-system. Here the guidance of a reference architecture is critical because the trade-offs in the architecture determine the scope that it covers. Complex systems always must deal with different life cycles, a good guidance is to assign only one life cycle to one component.

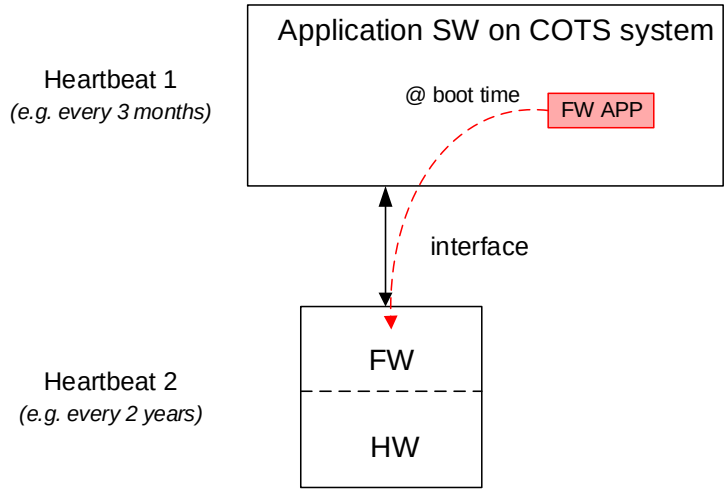

Figure 7: Improving flexibility of firmware update, by allocating the update facility to application SW

A typical (lower cost) option to deploy additional functionality to the market is by (automated) SW upgrades. Considering where to allocate necessary SW update facilities for the various system components can provide for more flexibility. As application SW typically will be updated more often than firmware of dedicated HW, planning flexibility of firmware application updates can be greatly improved if the FW update facility for the firmware/dedicated HW component is placed in the application SW (see Figure 7).

Interface management. The different life cycles and non-synchronized availability of system modules suggest interfaces between the various modules that have stability over the typical life span of the slowest cycling module.

Purposeful and systematical interface/API definition and management hence is important. Standardization of the major hardware/firmware/software interfaces provides for stability. Good practice is to have the interface determined by the **using** component (not the **providing** component). Interfaces of components with uncertain lifetime should be encapsulated rather than using their native interface directly.

A pattern for interface management to support life cycle variations is to provide capability interfaces on server-side modules (with longer development cycles). The applications (with shorter development cycles) then do not directly talk to the capability interface, but rather to an internal layer that represents the usage need (see Figure 8). Typically, the use case

need is not as broad as the capability of a server component. Interface changes from one server version to the other can then be managed in the service adapter, so that the application logic is not affected.

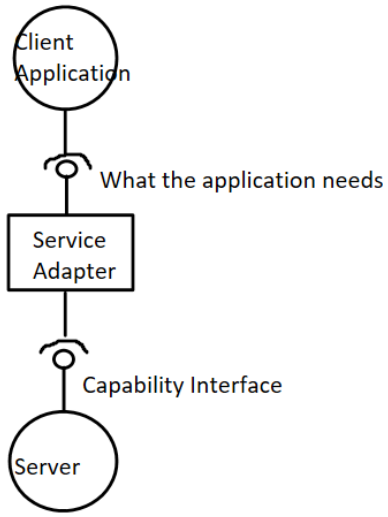

Figure 8: Server Adapter Layer interface pattern to decouple usage from capability

Tool support for interface management can help, work-arounds are adding a second (or extended) interface as a kind of "escape hatch". Finally, interfaces can be assigned life times too, and given expiration dates after which they are deprecated.

5.4 Fitting in new developments and unforeseen market needs

New developments and features to address unforeseen market need to be positioned into the (evolving) development roadmap. First, an impact analysis is should be done and corresponding concepts to be developed. The analysis must elaborate alternatives, review the partitioning, and identify the needed deltas for which components to arrive at the change details and corresponding concept description.

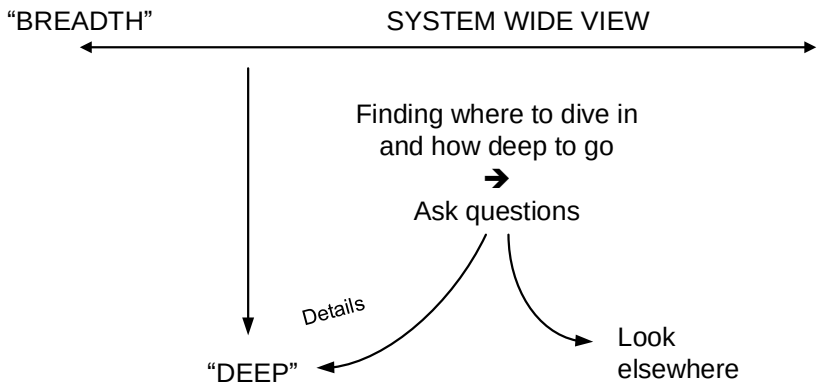

Figure 9: Checking deep or looking elsewhere in assessment of options

The default assumption is to use the "same architecture as last time". Only when a significant change is needed (in a non-functional, i.e. system quality) then a new architecture or update is to be considered. Assessment of options involves finding out where to dive in (and how deep to go), or to look elsewhere for a solution concept is a key balancing act (see Figure 9).

Final decisions are to be taken as late as possible (considering lead times, critical paths, costs), and on a need basis (schedule). For complex developments, building a demo / proto is useful to identify confirm the needed change, or to fail early when possible, such as to be able to reconsider the concept selection.

Alignment and harmonization. New requests and concept options should be compared/contrasted with already pending developments on the roadmap. Components on slow cycles require clustering and larger changes per cycle; components on faster cycles may operate on smaller, isolated changes per cycles.

Thus, a periodical update of the development roadmap, considering and weighing a set of changes at the same time, is prudent. Creating overview and alignment is a task for the architect or the system engineering group. For alignment of those change requests, identify components that change over time:

- Components that don't change need less attention.
- Perform impact analysis on components that do change.
- Acquire budget (alignment does not occur miraculously).

The slow life-cycle components identify the "must-have alignments". Architects present stakeholders with a plan for inclusion of a new change request, considering options and supporting data, as well as the architect's preference for the trade-off between development synergy and time-to-market. Shorter cycle times here reduce the "fear of missing out": the time delay penalty incurred by being moved out to the next cycle. Risks can be mitigated e.g. partial SW maturity / content; urgent updates can be achieved with an asynchronous SW update in the field.

When fitting in new developments and unforeseen market needs, architects should make conscious decisions on "go on" or "change because of" and provide supporting context/explanation of the bigger picture as of ongoing development activities towards stakeholders and customer project engineers.

6 Conclusions

In asynchronous development, customer project/systems development is decoupled from product platform and component development. In complex systems with multiple system levels, different life-cycle periods are inevitable. Life cycle then is a criterion for system partitioning. A trade-off to be made is between product cost efficiency and total cost of ownership over the life cycle. The origin or cause of different life cycles, i.e. a necessary change, are diverse: technology, module suppliers, faster time to market, development efficiency, changing or unknown market requirements, and market readiness.

Architecting for asynchronous development and lifecycles requires: an understanding of the future, both market and technology wise; planning ahead to align development steps and feature rollout with stakeholders; to manage system partitioning and interfacing and; to fit in new developments and unforeseen market needs as they arise.

A general strategy is to simplify complexity by good definition of building blocks (isolating sub-systems) with set of purposeful, systematically defined interfaces. A realistic, modular (platform) architecture, tailorable to various solutions is one of the success factors for asynchronous development. If systems operate on the edge of physics (e.g. electron microscopes, wafer scanners, MRI scanners), can these be designed as an asynchronous systems-of-systems: to what extent is that possible? On the other side of the spectrum, in mass volume products, hardware cost is paramount; every penny counts. This may lead to (unwanted?) close coupling to squeeze costs.

With respect to architecting for asynchronous development and life cycles, Product Line Engineering was a major step (van der Linden et.al., 2007). Since then, there has been little innovation in this field for architects overseeing complex systems: “hearing the same things for the last 10 years”. Architecting for asynchronous development and lifecycles is a challenging task indeed, relying on architecting knowledge, experience, and heavily dependent on people-centered processes such as road mapping and stakeholder alignment.

7 Literature

Evans, E. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

Leffingwell, D., Scaling software agility: best practices for large enterprises. Pearson Education, 2007.

Muller, G., How to Create a Manageable Platform Architecture, Gaudi site, 2005-2018 http://gaudisite.nl/HowtoManageablePlatformArchitecturePaper.pdf

Muller, G., Hole, E., Reference Architectures; Why, What and How, System Architecting Forum 2007 http://www.architectingforum.org/whitepapers/SAF_WhitePaper_2007_4.pdf

Muller, G., Hole, E., Roadmapping for a Changing World, System Architecting Forum 2015, (http://www.architectingforum.org/whitepapers/SAF_WhitePaper_2015_16.pdf)

Muller, G., The challenge of increasing heterogeneity in Systems of Systems for architecting, 13th Annual Conference on System of Systems Engineering (SoSE), Paris, 2018.

Phaal, R., Muller, G., An architectural framework for roadmapping: Towards visual strategy. Technological Forecasting and Social Change, 76(1), pp 39-49, 2009.

Scaled Agile Framework, https://www.scaledagileframework.com/

Shenhar, A., Dov D., Reinventing Project Management –The Diamond Approach to Successful Growth and Innovation, Harvard Business School Press, Boston, MA, 2005.

The Verge, “Tesla can change so much with over-the-air updates that it’s messing with some owners’ heads”, June 2018 https://www.theverge.com/2018/6/2/17413732/tesla-over-the-air-software-updates-brakes

Van der Linden, F., Schmid, K., & Rommes, E., The product line engineering approach, In Software Product Lines in Action (pp. 3-20), Springer, Berlin, Heidelberg, 2007.