embeddedworld2016
Exhibition&Conference
...it's a smarter world

Being Agile is not Enough

Joachim Pfeffer
peppair GmbH
joachim.pfeffer@peppair.com

Agile Approaches arose in the world of software development. Frequent releases provide early feedback with respect to requirements, technology and business value. Focused and empowered teams deliver huge amounts of software with very high quality. Transferring these concepts to the world of embedded development, the greatest challenge are frequent releases due to the interdependency of requirements. In this paper I will provide some concepts which help to understand and adapt agile approaches for embedded development and multi project environments.

I. Challenges for Agility in the Embedded World

Once hardware is involved, it gets hard to release new products in cycles of a few weeks. Normally it is not possible to develop, test and release single features, we always have to develop in certain batch sizes. While in agile software development the approaches focus on pure development, as build and test can be performed automatically, the production of samples and the test of an embedded system consumes a significant amount of time, which again causes slack time in the development team. This slack time normally is filled with other parallel projects.

Transaction time to release an embedded system may be reduced by paying express fees to suppliers and investing in test infrastructure. The return on invest is difficult to calculate. I will provide some approaches in the next chapter.

II. Economics of Product Development

A. *Management and Metrics*

What do we know about economics in product development? Which metrics do we measure in our development projects? Lean development guru Donald Reinertsen states that we measure proxy variables that are easy to measure but not useful for the project.

If you ask managers and developers about their motivation for process improvement, you may encounter some of the following statements:

- improve quality
- reduce team size
- reduce cycle time
- increase predictability
- increase efficiency

The next step is to ask how project profit would be influenced by these changes, not only the direction of the change in profit, especially the amount. Normally you won't get an answer.

A similar challenge is to measure the changes in profit in technical variables. What is the profit of, for example:

- reduction of controller cycle time by 1ms?
- 500h more MTBF?
- reduction of RAM consumption by 100kb?

Without having answers to these questions, it is very difficult to justify investments into the desired changes as for example 'cycle time'.

If you manage to express the value of the depicted aspects of process improvement in EUR, this will avoid unnecessary discussions and save money through targeted and measurable process improvement. Time to have a look at different aspects of cost in development projects.

B. *Cost of Delay (COD)*

While in production the 'lean' movement brought the following shift in mind: Inventory generates more cost than idle workers. Therefore, since the 1940s, production organizations recognize the cost of a workpiece that is not machined.

Development organization also have a kind of inventory: tasks. This inventory is not visible, thus normally cares about unprocessed tasks. Most managers in development

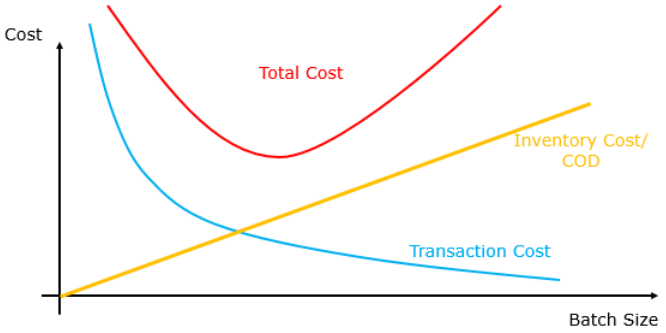
organizations don't have an idea which cost is generated, if a task is not processed for one day. I often get the reply, that untreated tasks do not generate cost at all. If this would be true, it would be best to expand projects to infinity.

Cost of delay (COD) exist but they are difficult to determine. When discussing economics of development projects the primary goal should be to define COD for certain set of tasks by asking: What would it cost, if we reach the milestone / deadline one day later?

Once COD is defined, efforts and investments for reducing cycle time can be justified. It then is also possible to determine the optimum cycle time instead of seeking the shortest possible cycle time. Even in cases where the dimension of COD cannot be determined yet, recognizing the presence of COD can help to trigger necessary changes in processes, workflow and equipment.

C. *Transaction Cost and Batch Size*

Like in production, total cost in development consists of inventory cost (i.e. COD) and transaction cost for a release. Total cost is the well known U-curve diagram



Although batches are normally associated with production, most of our development processes also use batches. Some batches are necessary because the type of the product, when designing a PCB, a certain set of requirements (batch) is processed. Some batches are necessary because of the infrastructure and equipment, for example when using test benches. The third case are batches which are forced by the process, the worst case are the so called 'stage gated' processes which use the maximum possible batch size which is according to the diagram above a very unfavorable 'operating point'.

In other words, when using batches, tasks have to wait for the completion of other tasks and cause cost of delay. Batches delay feedback on requirements and quality and increase risk.

On the other hand, the diagram above also shows that too small batches are also expensive. Small batches reduce COD but will only reduce total cost if transaction cost can also be reduced.

D. *Development Cost and Agile Approaches*

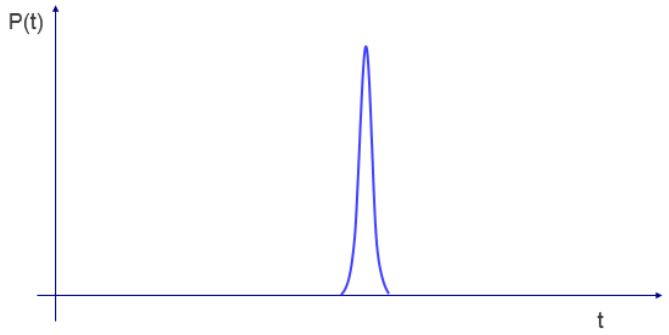
Agile approaches use small batch sizes (Scrum) or even single piece flow. They recognize implicitly the impact of COD, especially the benefit of early feedback. Agile approaches accept the variability in product development by focusing on frequent releases and re-planning.

Agile approaches do not provide metrics in terms of project cost and project profit, they improve batch sizes and cycle time empirically. The awareness of the correlations depicted above can help to reduce cost in embedded development. If COD for a certain project or certain tasks are defined, it is even possible to define best batch sizes mathematically. Still the greatest impact on COD is cycle time, which we will discuss, in the next chapter.

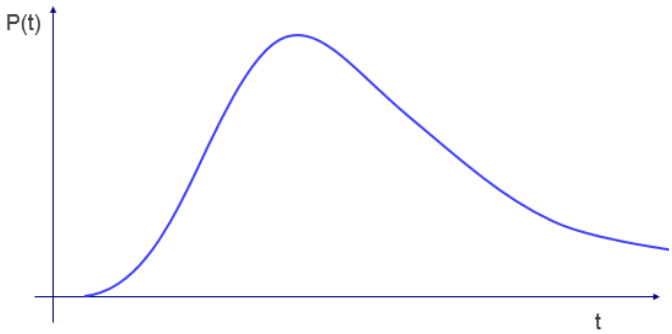
III. QUEUE AND BACKLOG MANAGEMENT

A. *Variability*

As I stated above, agile approaches accept the lack of predictability in product development. In production, processing time for a certain process step is very predictable; it has a narrow probability distribution:



Whereas probability distribution in product development is much broader due to innovation, technical challenges etc.:



This is why 'production thinking' approaches like huge Gantt charts for several years does not work for product development. Agile approaches use very small increments to 'reset' variability periodically.

B. *Queueing Theory*

If production thinking is not useful for product development with regard to variability, are there any other concepts that might be useful?

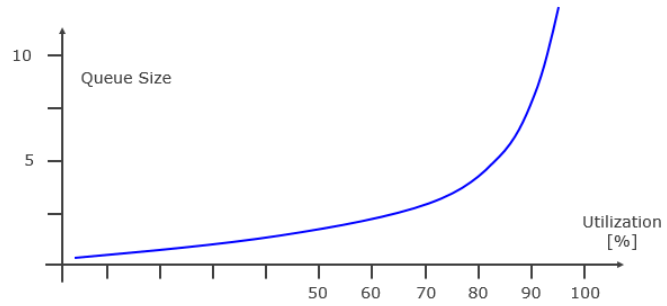
At the beginning of the 20th century mathematicians developed models to describe the behavior of an environment which has to handle an unpredictable processing time as well as an unpredictable arrival rate, the telecommunication industry.

The most simple queue concept is a single server (in our context a development team, an expert, a measurement lab etc.) and a single queue of tasks for the server. For this setup, the

utilization of the server (ρ) can be set in relation to the que size L with the following simple formula:

$$L = 1 / (1- \rho^2)$$

For the further discussion, I will show this relationship in a diagram:



What can we see in this diagram? If the utilization (efficiency) goes up to 100%, the queue size gets infinite. Many organizations try to utilize their developers by 80%, which seems to be a pretty good operation point, before the slope of the curve gets too steep. However, operating a team at 80% would mean to have an average queue size of 3.2 items. Regarding real queue sizes in development organizations we can see, that most developers are utilized by over 98%.

Queue size is linear to lead time, so the queue size determines cost of delay (COD). In other word: queues in development contain expensive inventory. Thus the strive for high efficiency generates huge costs. Good flow and throughput in development require a utilization lower than 100%, thus a certain amount of slack time.

The best operating point for an organization can be determined mathematically, if cost of bad utilization (easy) and cost of delay (hard) are known. But also without knowing the current COD, the diagram above shows, that there are two operating points which are the worst cases: 0% utilization and 100% utilization.

The good news: As the slope is very steep at the right border of the diagram, in environments of high utilization even a small increase of capacity causes a significant reduction of queue size and thus cost.

C. *Queues and Backlogs*

Agile approaches create slack time (i.e. buffers) intentionally to keep the tasks flowing. Scrum limits the sprint scope and in Kanban environments the pull systems reduces the average utilization. Both approaches value flow and cycle time more than efficiency. The reason is: cost of delay.

The concepts of the queuing theory can be used to manage backlogs (which are queues). Queue size is the earliest possible indicator of congestions and over-utilization. Cycle time however gives a very late indication. Despite that, most organizations measure cycle time instead of queue size.

The checkout staffing in supermarkets is a good example for queue management. No supermarket measures the cycle time for the customers. Checkout staffing is based on the visible queue size. This principle can be also used in embedded development. Not only development teams are fed by queues, also centralized resources like measurement labs, test benches or experts have queues. A good approach is to give each resource its own backlog and to define a threshold on which additional resources are involved. For example: “If the backlog of our layout team excess 10 items, we give further layout jobs to our service contractor outside of the company.”

Summary: Agile processes start already at the backlog. Queuing theory is a useful model to set up rules and procedures to detect congestions and to manage the backlog actively in order to increase flow.

D. *Example Calculation*

Here I provide an example calculation for a measurement lab which is the constraint of the company:

- Measurement jobs are managed in a tool or a backlog => continuous measurement of queue size
- Average queue size L=80 jobs
- Average processing rate (cycle time) $\mu = 1$ job/day
- Calculation of current lead time: $T = L / \mu = 80$ days (working days)

Now the management decides to reduce the current lead time to 5(!) days in order to keep the development projects running. What would be the desired capacity?

- Step 1: Calculation of current utilization (L=80) $\rho = 98.7803064$ %
- Step 2: Calculation of current arrival rate $\lambda = \mu * \rho = 0.987803064$ jobs/ day
- Step 3: Calculation of the utilization for L = 5 $\rho = 85.4101966$%
- Step 4: Calculation of the required capacity ($\rho = 85.4101966$%)
  $\mu = \lambda / \rho = 1.156539972$ jobs/ day
- → increase by 15%

Summary: Increasing capacity by 15% will reduce the average lead time from 16 weeks to one week! (this is because of the mentioned steep slope of the diagram at the right border).

IV. MULTI PROJECT ENVIRONMENTS

A. *Challenges*

Normally development organizations create products in a multi project environment. Even if the advantages of a focused single project approach are recognized, it is often necessary to deal with several projects within a development team due to the processing time for sample setup (layout, sample shop etc.) and test.

With multi-project environments, agile teams encounter a problem that is not present in ‘clean agility’: prioritization between projects. Core agile approaches like Scrum or Kanban do not provide concepts how to handle several projects. A very useful approach is the ‘critical chain project management’

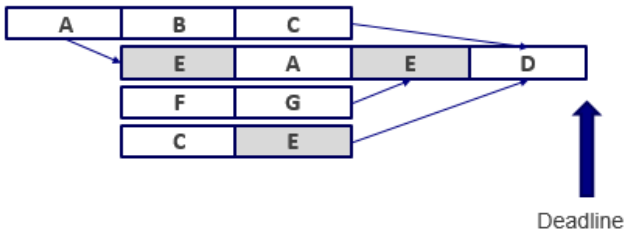
(CCPM), introduced by Eliyahu Goldratt in the 1980s. CCPM, as well as agile approaches, does not manage timelines. Instead both 'worlds' manage the sequence of tasks and create a floating timeline. However, in real life projects have serve certain milestones and deadlines in time. CCPM provides concepts to aim at hard milestones using a floating timeline, thus CCPM is a perfect 'add-on' to agile approaches.

B. Critical Chain Project Management (CCPM)

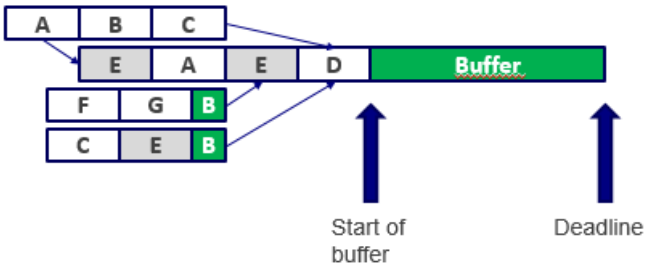
When a project schedule is set up (most of us use Gantt charts), someone will ask developers for their estimations. As in most organizations there is no difference between estimation and commitment, developers keep their actual estimation secret and add a buffer into the official estimation (commitment). Therefore, these schedules contain many secret buffers, often 50% or more time in a schedule is a buffer.

When the projects starts, the secret buffers vanish again, mainly due to the following effects: On the one hand, developers know about their buffer which influences their personal time management and the buffers diffuse into space. On the other hand, if the execution of the task matches the estimation, the developer can't deliver the task results, otherwise his secret buffer would get disclosed.

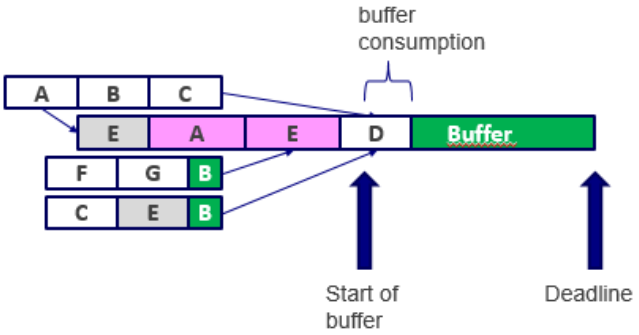
This is why we normally get a schedule with invisible unmanaged buffers:



CCPM takes another approach. CCPM removes secret buffers from the work packages. If the organizational culture is not yet ready to provide net estimations, all available estimations are reduced by 50%. Thereafter the removed buffer is added to the end of the critical chain of the schedule to meet the deadline again. This is a visible manageable buffer:



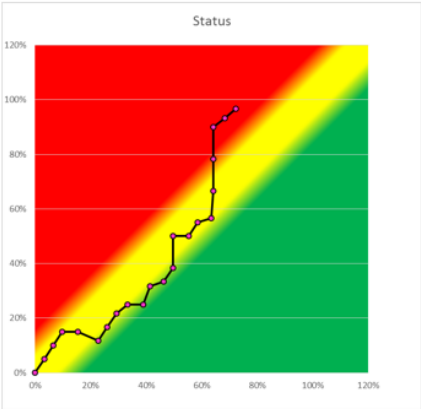
As the diagram above shows, supplier paths are also buffered to protect the critical chain. What will happen, if a work package needs more time than the net estimation? Everyone in the project is allowed to use a part of the common buffer. So every package takes the time it really needs to complete it with the desired quality. No one charges the one who exceeds his estimation (this requires a cultural change). Thus, the schedule floats into the buffer. In the following example the pink packages exceeded their estimation and shift the schedule into the buffer:



As the buffer size and the buffer consumption are known, there is a simple metric to monitor the project:

Project Health Index = Progress / Buffer Consumption

CCPM displays continuous measurement of the health index in the so-called fever curve. Here an example:



In an ideal world, the buffer consumption would increase proportional to the progress on the critical chain. If the index drops below one, the buffer consumption is too high in relation to the progress which requires immediate action to get the index back above one. This gives a transparent and early information for early corrections and high predictability, in contrast to the classical approach with runaway buffers and the unfounded hope that a delay can be caught up by the following work packages.

C. CCPM and Agile Approaches

Backlogs in agile approaches represent the critical chain. In distributed development, every department / team / expert has his own backlog (supplier paths are also buffered). If the mentioned resources have to switch between several projects, the health index is the key metric for prioritization: Process tasks of projects / subprojects with the lowest index first.

V. SUMMARY

A. *Putting it all together*

Development organizations are full of invisible inventory: tasks. Tasks that are not processed create cost for the project. Knowing these costs of delay (COD) is essential for all decisions in the project and for driving improvements in the organization, independent if the project works agile or classical.

Recognizing the impact of COD, the utilization of teams and resources has to be adjusted to reduce cost. In most cases utilization has to be decreased to provide agility and flexibility. Concepts like Scrum or Kanban have built-in methods to reduce utilization. Resources which do not work agile like layout departments, measurement labs or experts have to reduce their utilization by an active backlog management. The supermarket principle of controlling resources by queue length is a powerful tool, once queue size is measured.

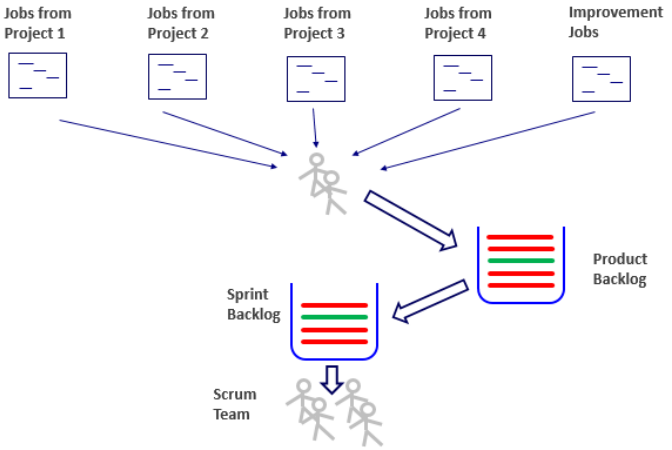
Embedded development has to handle several projects at the same time, not least because of the time required to build and test a sample or a prototype. This adds a new challenge for agile organizations which are normally focused on one project: prioritization between projects.

Agile approaches do not provide concepts for the prioritization between projects and due to their floating timeline approach. They do not provide methods to target hard deadlines. Critical Chain Project Management (CCPM) provides a transparent buffer management for floating timelines and matches perfectly to agile approaches which came up 20 years later on.

B. *Example*

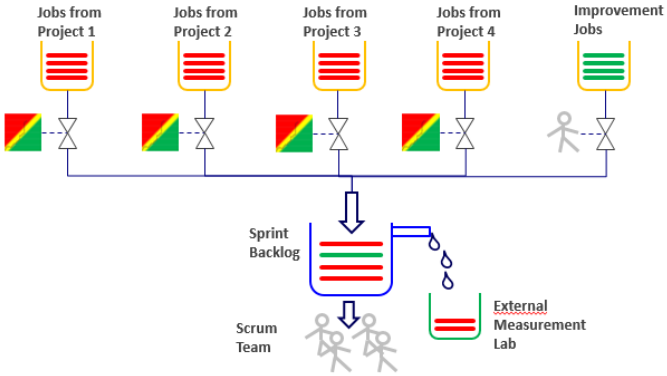
A good example for the combination of these concepts is a case study from my consulting business:

A centralized measurement and development team, which had to provide measurements and redesigns to several customer projects ran into capacity problems when the number of projects to serve increased continuously. The workload for managing and prioritizing the deliveries for up to 100 sub-projects got very high and so the team decided to switch to Scrum to get more transparency:



Despite the new transparency, the workload remained very high. The goal was to get a metric for the prioritization and to set up rules for the management of resources to assert a certain lead-time for the projects.

The first step was to split up the backlog into project-specific ‘feeder backlogs’. The feeder backlogs are the floating timeline and are monitored with a CCPM fever curve to match the due date. Projects with the lowest health index shift the next items into the team backlog. The team backlog itself was monitored (queue size) to provide a certain lead-time. If the backlog size exceeds the threshold (determined by COD), measurement jobs are shifted to an external measurement lab.



VI. BIBLIOGRAPHY

[1] Donald Reinertsen, „The Principles of Product Development Flow “, 2009
[2] Uwe Techt, Holger Lörz, “Critical Chain”, 2011