

Lean System Integration at Hewlett-Packard

Kathy Iberle
kiberle@kiberle.com

Abstract

Discover how HP is applying Lean principles to drive the integration of large systems, resulting in both higher quality and higher productivity.

In the HP printer business, Lean integration is

- making complex programs easier to manage by providing visibility into what the product can and cannot do at any point in the development
- improving the customer's experience by making customer workflows functional and visible early and often throughout the lifecycle
- reducing cost by driving synchronization of delivery across technology components

This paper provides an introduction to the methods known as Lean Software and Systems Development. Lean is able to handle situations which are difficult to handle using the most commonly known agile methods, such as large, complex, and partially waterfall systems, by applying methods deriving from queuing theory and statistics.

Lean methods are demonstrated in this paper with examples and results from actual projects in the HP Inkjet and LaserJet businesses.

Biography

Kathy Iberle is a senior software quality engineer at Hewlett-Packard, currently working at the HP site in Boise, Idaho. Over the past twenty-five years, she has been involved in software development and testing for products ranging from medical test result management systems to printer drivers to Internet applications. Kathy has worked extensively on training new test engineers, researching appropriate development and test methodologies for different situations, and developing processes for effective and efficient requirements management and software testing.

Kathy has an M.S. in Computer Science from the University of Washington and an excessive collection of degrees in Chemistry from the University of Washington and the University of Michigan.

Copyright Hewlett-Packard, 2010

Acknowledgements:

A big thank you to Dr. Laurian Dinca and to Chris Moehring, for providing the data and project history which was included in this paper.

First published at the Pacific Northwest Software Quality Conference 2010

Introduction

What is Lean Product Development?

Lean Product Development is a paradigm for developing products which views the development organization as a system or machine to produce profit. Lean Product Development uses proven engineering, mathematical, and statistical methods to maximize the system's output of profit by engineering a rapid, smooth flow of saleable features and capabilities through the development organization. This is done by

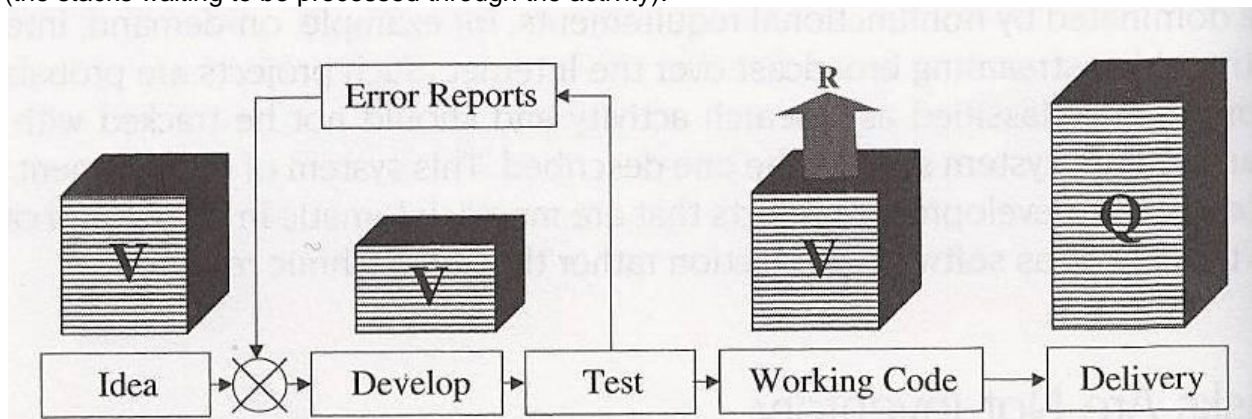
1. Modeling the work of the organization as a system to turn ideas into money. The system model consists of activities and wait states or *queues*.
2. Observing that unfinished work isn't free - there is money tied up in it.
3. Applying a body of mathematical knowledge known as *queuing theory* to optimize the profitability of the system. This optimization requires
 - o minimizing the money tied up in unfinished work
 - o minimizing the operating expense
 - o maximizing the output of saleable product

Let's look at these ideas one at a time.

First, what does the work look like when modeled as a system for turning ideas into money?

Lean Product Development models the development system as items moving through a series of activities and wait states. Each activity takes more than zero time. Whenever items have to wait to be processed through an activity, a line or *queue* forms, where items wait to be served.

Here's a very simple model of software development, showing four activities (the boxes) and four queues (the stacks waiting to be processed through the activity).



(Anderson 2004, p. 53)

In this model, we can measure several things

- *Throughput* = stuff of value to the organization. "Value" means it is ready to be sold, or used to reduce internal costs, or otherwise immediately provide concrete value.
- *Investment* = money invested in material waiting to start, in process through the system, or waiting to exit from the system.
- *Operating expense* = money spent to run the system

In order to measure anything at all, there must be discrete chunks moving through this system. In a traditional waterfall development model, the entire body of work moves through the system as a single large chunk. In Lean Product Development, the work is broken into smaller chunks which move through the system sequentially, one after another.

A key difference between Lean Product Development and waterfall is that Lean Product Development insists that the chunks moving through the development system **must** be saleable features, **not** tasks. This ensures that all measurements and analysis of the system will focus on maximizing the throughput of *saleable features and capabilities* (and thus profit) rather than the throughput of *tasks*.¹

This will sound very familiar to anyone who has worked with agile development. I'll explain the difference between agile development and Lean Product Development in a bit.

Second, why does unfinished work cost money?

The cost of unfinished work is easier to see in an example using physical objects. Let's look at a manufacturing line.

On a manufacturing line, the system consists of a number of assembly steps, each performed at a different assembly station. The partially completed items move from one assembly station to the next. At each station, some parts are added to the partially-assembled item.

In the old days, it was assumed that maximum output was achieved by having every assembly station utilized at 100% capacity - that is, running full tilt all the time. This required having enough parts on hand to keep every station running regardless of what was happening at other assembly stations.

This makes sense, except for one thing: the different assembly stations *don't* run at the same speed all the time. The variation between stations has two results:

- Each assembly station needs a stockpile of parts large enough to handle the fastest possible input.
- Partially completed items pile up in a queue in front of the slower stations.

The stockpile of parts and the partially completed items both contain parts not yet part of a saleable product. This is the *inventory*. The inventory ties up money in several ways.

- Money invested in purchasing the parts
- Owning and maintaining shelving & storage space for parts
- Maintaining an elaborate tracking system and related processes to keep track of the multitudes of parts
- Time spent by people to communicate with each other and the tracking system concerning the status and location of the parts

Just-in-time (JIT) manufacturing practices were introduced in the 1980s to minimize this investment. JIT focuses on finishing items from start to end as rapidly as possible rather than utilizing every assembly station 100%. Basically, the whole system runs at the speed of the slowest assembly station. Parts are pulled into a workstation just before they are needed, rather than stockpiling parts. This means there are smaller stockpiles and fewer partial assemblies sitting in queues. Using JIT, it was often possible to reduce the number of parts on hand at any given time by 80% or more. [Reinertsen 1997]

Because there are fewer parts sitting around, the shelf space needed was vastly reduced. The tracking systems became much simpler, sometimes just a set of index cards instead of a ledger-based or electronic system. The total amount of communication needed was reduced. All this simplification not only made life easier, it also freed up huge amounts of cash which could be invested in other parts of the business.

In software development, we also have an inventory of material which is not yet saleable. The agile community has recognized from its start that this inventory likewise ties up money and resources. Software development's inventory consists of ideas, specifications, design documents, code fragments,

¹ In the case of development for in-house use, "saleable features" are replaced by "features which reduce labor or cost within the business". The focus is always on providing usable value.

untested code, defect reports, etc. Just like parts in a manufacturing process, the documents and code produce no profit until the final product is finished and shipped. The software inventory is not as obvious because it physically exists as bits on a drive rather than parts on a shelf, but the costs are much the same:

- The labor invested in developing the ideas, documents, and code
- Owning and maintaining server space for storage
- Maintaining configuration management, requirements management, and defect management databases to track the location and status of the multitudes of pieces.

In addition, unlike the physical parts in a manufacturing process, the half-finished software inventory usually loses value over time. The environment changes, the customers' needs change, or the developers forget the details & have to reinvestigate. If a long enough time passes, the whole thing has to be thrown away and done over.

Third, how can queuing theory help us optimize our development system?

There is a substantial body of mathematics known as *queuing theory* which accurately predicts the behavior of a system of activities & queues under various conditions – high-capacity activities, low-capacity activities, big batches of items, small batches of items, variability in the “size” of the items (the amount of resources required to process the item), and so forth. Once we've split our work into chunks, and modeled our system as activities & queues, we can apply queuing theory to speed up the throughput of the chunks through the system.

A set of chunks of work will usually move through a system fastest when the chunks are started off one at a time (or a few at a time), rather than all at once. The number of chunks in the system at any given time is known as the *work in process* or WIP. Keeping WIP low reduces the amount of tracking and communication needed, just as a smaller number of parts in inventory reduces the amount of tracking needed.

Low WIP has one other very helpful effect – it reduces the amount of task-switching. A human being cannot switch from one task or topic to another for free. The simplest switches take a few seconds. In a switch between complex tasks, the worker may take as much as ten minutes to come fully up to speed on the second task. The time spent task-switching is mostly wasted time. When we have multiple projects in process at the same time across multiple people, the cumulative cost of task-switching and the additional communication needed is huge. Cutting down the WIP is like getting free money, or more accurately, free hours of work.

The traditional waterfall process of setting all features into motion in lockstep produces the highest possible WIP for the workers. The WIP is much lower when a set of features is designed, coded, tested, and fixed to completion before starting the next set.

Agile and Lean Product Development Compared

Agile development also recognizes the value of splitting work into small, saleable chunks, and minimizing the amount of unfinished work in progress at any given time, and minimizing task-switching. The main difference between agile and Lean Product Development is that most of the agile community explains why agile works in terms of heuristics or rules-of-thumb such as “the last responsible moment”, whereas Lean Product Development explains why Lean works in terms of a system model and queuing theory. Lean Product Development presents the queuing theory algorithm and shows the user how to work out the answer for a type of situation when it appears in their particular system.

Most agile development principles or heuristics are the answer derived when queuing theory is applied to a small, relatively simple development system. They are correct, for that particular type of system. However, when the system gets larger and more complex, application of queuing theory often leads to a different answer. This allows Lean Product Development to address large, complex development

systems clearly and systematically, whereas the agile methods often struggle with these systems. It also allows Lean Product Development to be used in mixed systems, where part of the system is still waterfall.

A good example of this is the practice of Continuous Integration. Agile and Lean both agree that a short time between making an error and receiving feedback on the error will reduce the time needed to diagnose and solve the error. A developer will more quickly find and fix a problem in code he wrote a few days ago as opposed to code written a few weeks or a few months ago, because there is less time spent remembering or re-learning the code.

Since a short feedback loop saves time, should we make the feedback loop as short as we possibly can? In *Extreme Programming Installed* [Jeffries et. al. 2001, p. 78], the authors say “The extreme solution, of course, is to integrate as often as possible. We call it *continuous integration*. A good XP team will integrate and test the entire system many times per day. Yes, many times per day.” The authors follow this with an observation that this is impractical in many development systems because the build takes too long, and exhort the reader to reduce build time to near zero. Continuous Integration is a heuristic.

In contrast, Lean looks at the economics of the entire system, comparing cost and value. Agile and Lean both agree that earlier feedback adds value. Lean asks more questions: How much value is added? How does that value vary over time? What does it cost to add that value?

Feedback on errors has an inherent value to the developer – I need to know about the error. There’s also some additional value to me if I find out about the error before I forget the context, because I don’t have to re-learn the context and this saves me some time. In my experience as a developer, this additional value of fast feedback also changed depending on the type of error reporting. The value of feedback on coding errors dropped off quickly over a matter of hours, reaching zero about three days after I wrote the code. After that, it didn’t much matter whether I got the feedback on day 4 or day 24 – I’d forgotten the exact logic. This is why compiling and unit testing immediately is so important. The additional value of a design error dropped off more slowly, such that feedback within three or four days was nearly as good as feedback on the first day after writing the code.

Now let’s look at the cost of getting this quick feedback. Earlier feedback is usually achieved by getting feedback more frequently. This means the developer has to submit code into integration more frequently. Depending on what the integration & subsequent feedback consists of, this submission may not be zero cost.

For instance, suppose the developer is required to run a set of check-in tests before checking in the code. Routinely getting feedback on day N+1 requires checking in every day, which in turn means running those check-in tests every day. This cost is not zero. So, the additional value to the developer of quick feedback is offset by the cost of the check-in itself. At some point, the check-in becomes more expensive than the value of the feedback. After that making the integrations closer together doesn’t improve the overall system – it makes the overall system worse.

The exact point at which this happens will be different for different organizations, depending on the cost of the check-in, and the additional value of the quick feedback. The Lean approach to this problem is to

- 1) Find the approximate optimal point and integrate that often – but not more often
- 2) If the length of the feedback loop is still a significant problem for the overall system, work to reduce it.

As the Poppendiecks say [Poppendieck 2007, p. 202], “attack the set-up time and drive it down to the point where continuous integration is fast and painless.”

The tools of Lean Development allow you to look at a system where the cost of checking-in isn’t zero, or the feedback is time-consuming to interpret, or any number of other non-ideal situations, and figure out where the sweet spot is – the point at which overall development cost is lowest. These are very powerful tools.

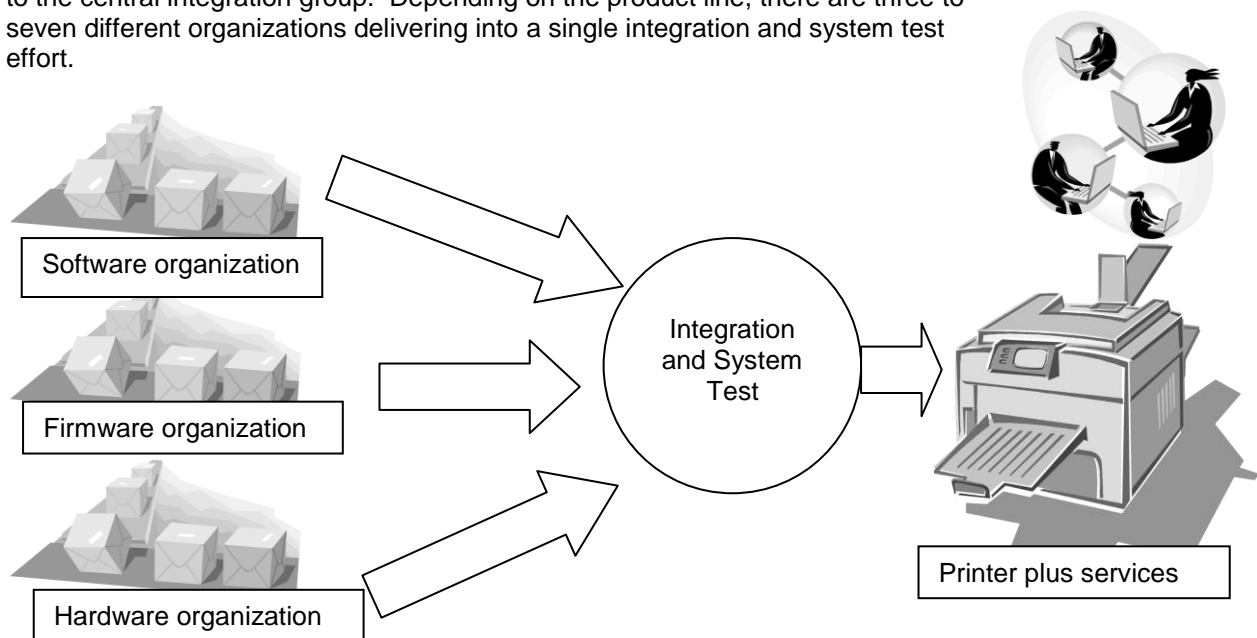
Lean Development Applied – Product Integration at HP

Let's look at a real-life application of Lean Product Development to a system which is large, complex, and in some ways downright messy. This is the story of our organization's application of Lean Product Development, and the benefits we have derived to date.

Starting Point – Agile Development of Subsystems, Waterfall Development of Systems

Our organization was formed in early 2009 as an integration and test group serving both Inkjets and LaserJets. Hewlett-Packard (HP) releases a large number of Inkjet and LaserJet products each year. These products consist of a great deal of sophisticated software, firmware, hardware, and allied web services, which are produced by dozens of individual teams. The teams are organized into several large groups, each of which delivers a major subsystem which will be incorporated into multiple products.² Most of the groups are now using some form of agile or incremental delivery.

The end result of this is much like a set of conveyor belts, where each organization is delivering features to the central integration group. Depending on the product line, there are three to seven different organizations delivering into a single integration and system test effort.



Despite the fact that each of the subsystem organizations is independently using some form of agile development, the product level development still felt very “waterfall”:

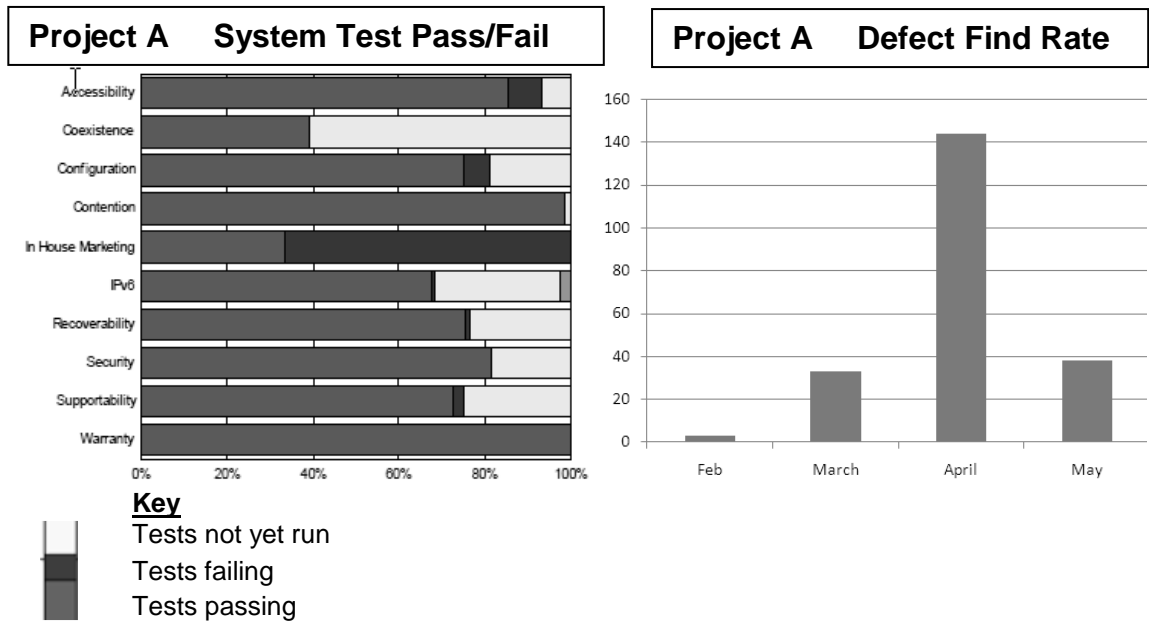
- System test started quite late. Most features were not testable earlier than “code complete” because one organization or another had not yet submitted their piece of the feature.
- Subsystem organizations would receive defects from system test on some code well after the code had been “finished” from the subsystem’s perspective
- Final release schedule was rather unpredictable. The progress of individual organizations did not accurately predict the progress of the overall program.
- There was also a hidden cost – the cost of fixing defects was higher than strictly necessary, due to the length of time between releasing code to integration, and the eventual receipt of defect reports from system testing.

² Developing shared software which is configured for individual products is known as *Software Product Lines* or *Software Product Family* style of development. This is a large and interesting topic, but outside the scope of this paper. See the reference list for references, or Google “software product lines”.

Let's look at Project A, a typical high-end LaserJet project prior to the introduction of Lean Product Development.

System testing consisted of running a large number of tests which examined end-to-end functionality of the system. The tests were organized by capabilities or quality attributes, such as security, compatibility, recoverability, coexistence, and so forth. The organization of the tests and the test execution plan both assumed that all the subsystems would all be delivered to system test at once, with all features functional. There was no easy way to identify tests for given features and run those earlier or later than the whole.

Typical reporting from system test included various representations of the defect find rate (shown to the right) and snapshots of system test pass/fail status (shown to the left).



System test ran effectively for only three months of a much longer project (typically twelve months or so), and the bulk of defects found by system test were found in a single month near the end of the project. This created an extremely long feedback loop for code written near the start of the project.

The traditional reporting for system test measured progress at passing a standard set of capability-based tests. This didn't tell the Marketing department anything about the status of specific new features.

The newly formed system test and integration organization wanted to start system test earlier and reduce the length of the feedback loop. However, other groups had tried this in the past with limited success. A new approach was needed. The system test organization decided to attempt Lean Product Development at the system level.

Let's look at how we did this and what happened.

Breaking the Work into Chunks – the Slivers

The first step in applying Lean Development principles is to break the work into **chunks with business value**.

Customers don't purchase tests. They purchase features and capabilities which can do work for them. The first big paradigm shift for this group was moving from focusing entirely on *tests* to focusing on *features and capabilities*. This meant we needed to re-organize our tests around the features and abilities in which the user was interested. The tests were already organized by capability, but the feature aspect was missing.

Essentially, we were looking for requirements. Most of the subsystem organizations were using "user stories" and/or lists of functional requirements, either in spreadsheets or in databases. We first looked at those data stores to see if we could identify useful "chunks" at a system level. "Useful" meant:

- saleable (have clear and understandable value to a user)
- applicable at a system level – not specific to one subsystem
- broad enough to avoid drowning us in millions of chunks & the accompanying overhead

We found that, between the multiplicity of technologies in use, and the lack of a common method for identifying and storing *system-level* requirements as opposed to subsystem requirements, we could not easily identify a single consistent set of system-level requirements. We also observed that well-written system-level requirements could be generic enough to be applicable to multiple related products, especially when a user scenario was involved.

We concluded that we could best drive our test re-organization by writing our own high-level requirements, leveraging off of existing system-level requirements whenever possible. We decided to basically take the idea of a user story up one level, and define the "chunks" as a set of user scenarios focused on a particular type of customer doing a particular type of work. The chunks are known as *slivers*.

A typical set of slivers is shown to the right, for the feature area "Printing".

The first few slivers are "getting-started" slivers:

- Minimum entry criteria for this feature area: for instance, can the system print at all?
- Integration slivers (usually named "framework"): do the various subsystems talk to each other successfully? These are often gray-box tests.

The rest of the slivers in this example are user scenario slivers, as suggested by their names. Those usually comprise the bulk of the slivers.

Each sliver contains in its definition:

- A description of the customer
- Specific things the customer wants to do
- How we would know if the customer was satisfied



Here's the definition in a typical User Scenario sliver.

Title: Casual Photo Printing

Customer Description:

A user who casually prints photo content in a home or office using common photo printing applications. Typical usage is on standard and custom photo types and sizes. The number of users in this sliver can range from one to five. These users can exist on mixed OS platforms (i.e.; Mac, Windows) and varied connectivities (i.e.; USB, Wired, Wireless). These users typically exist in an unmanaged environment.

As a User I would like to print:

1. 4x6, 5x7, and 8x10 photo prints
2. Day-to-day family/ friends pictures
3. Scrapbooking
4. Facebook - Social networking sites
5. Snapfish - Online picture storage

Customer Satisfier:

1. My photo jobs print right the first time!
2. Image quality looks comparable to what user sees on pc.
3. Image quality is consistent over time.
4. Draft quality provides useful output.

You may notice that the sliver isn't very specific about the product under test. That's because the slivers are designed to be reusable across related products. The integration and system test group typically has around 20 different devices in test at any given time, most of which share large sets of features with each other. In addition, there are many user scenarios which are common to large groups of printers, particularly within individual product lines. We decided to take advantage of the enormous overlap between individual printers by defining reusable slivers whenever possible.

Once the slivers were defined, we had chunks of a reasonable size with clear user value. Next, we needed the ability to measure completion of the chunks, which means we needed completion criteria.

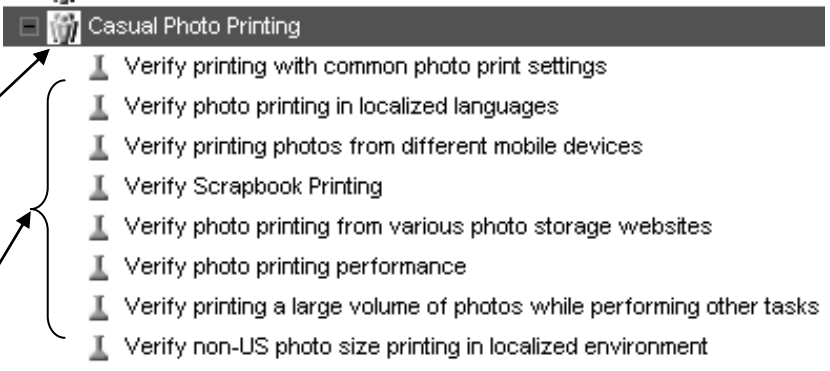
The completion of our slivers is defined as "passing system test"

- A sliver cannot enter system test until all involved subsystems have delivered a version of their subsystem which supports this sliver.
- A sliver cannot enter system test until all subsystems are passing earlier levels of testing at subsystem levels.
- System test starts with a simple entrance test to verify that the subsystems have been successfully integrated and are at a reasonable level of quality.
- The rest of system testing consists of running the tests meant to check the behavior specified in the slivers.

Since the completion criteria included running tests, we needed tests!

This is where it becomes very useful to have an integrated requirements-test management system, such as HP's QualityCenter. We wrote our sliver definitions and our test design into the requirements module of QualityCenter.

Each sliver consists of:

- A use-case type requirement which contains the definition of a *user scenario* (a description of expected user activities), as shown on the previous page.
 - A set of child test requirements stating how the system shall be tested for compliance with the use-case requirement.
 - The actual tests (not shown)
- 
- The diagram shows a dark grey box at the top with a small icon and the text 'Casual Photo Printing'. Below this box is a list of nine test requirements, each preceded by a person icon. Two arrows originate from the first two bullet points of the text to the left. One arrow points from the first bullet point to the 'Casual Photo Printing' box. The other arrow points from the second bullet point to a bracket that encompasses the entire list of test requirements.
- Verify printing with common photo print settings
 - Verify photo printing in localized languages
 - Verify printing photos from different mobile devices
 - Verify Scrapbook Printing
 - Verify photo printing from various photo storage websites
 - Verify photo printing performance
 - Verify printing a large volume of photos while performing other tasks
 - Verify non-US photo size printing in localized environment

The test requirement titles are simple statements of what shall be tested. The body of the test requirement (not shown) contains more detailed test design. Reading the titles is a very quick and powerful way to understand the test coverage, while putting the details of the test design in the body of the test requirement means we have all the test design information in one place.

The tests are then written and linked to the test requirements, so we can use the test requirements to index the tests. It is also possible to classify the slivers and the test requirements several different ways, so one can examine the coverage from different perspectives. The slivers and their child test requirements are organized by user activity to maintain a focus on users, but the test requirements are also indexed by quality attributes (performance, reliability, etc) so we can assess coverage in this dimension.

Our tests had previously been stored in home-grown test management systems, which stored only the tests. When we converted our tests from their earlier test-only format into this requirement-and-test format, we reduced the number tests by an average of 25% *without reducing coverage at all*, simply because we were able to spot duplication of coverage and outmoded tests so easily. Other HP organizations have had similar results.

Now we had chunks which had user-value, and we had a way of detecting when a chunk was done. We were ready to start Lean Product Development – running the chunks through a development system.

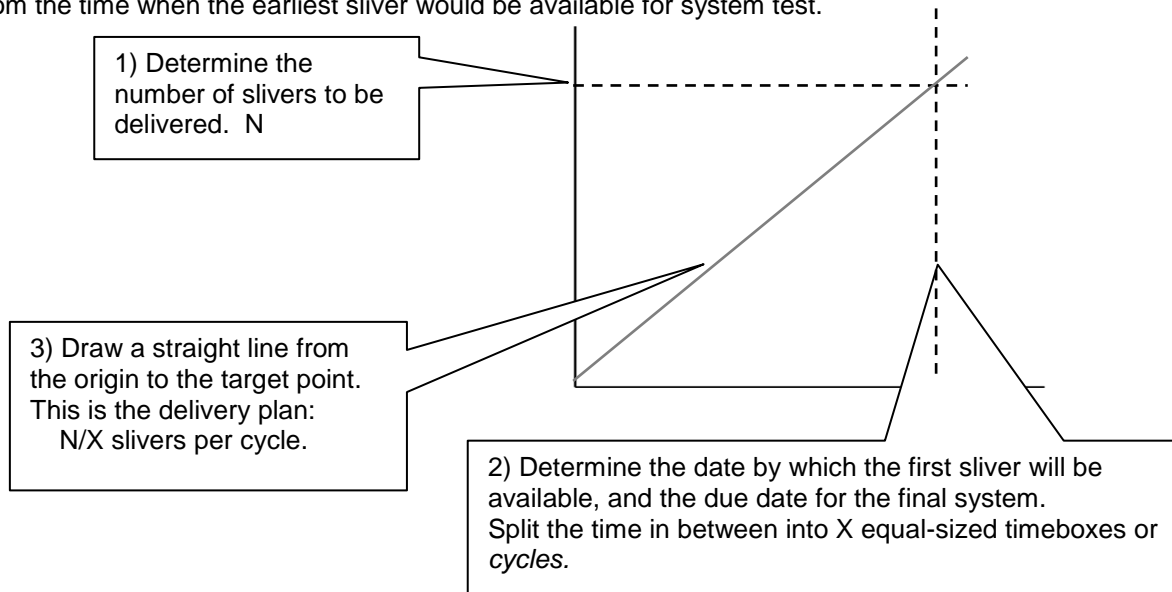
Setting up a Schedule – Session Planning

Now that we had the work broken into slivers, Lean and agile principles both suggest that the least expensive way to deliver the slivers would be to deliver them gradually throughout the program, rather than all at once near the end.

Most of the subsystem teams were already doing some form of agile, so the idea of delivering in chunks wasn't foreign to them. None of them had the same set of chunks, however. We presented our slivers as a reasonable *system-level* set of chunks, which would be common to all organizations during system test even though they still had their own individual chunks during component and subsystem development. This meant each organization had to create some sort of mapping between their chunks of development and our slivers. We had around a hundred slivers at this point, many of which were almost entirely legacy features.

The Marketing and Customer Satisfaction team members usually found that the slivers directly represented the features in which they were interested, and were eager to see if this method would lead to reporting progress of individual features.

We presented the simplest possible plan to the development teams: a linear delivery of slivers, starting from the time when the earliest sliver would be available for system test.



The different organizations had their own different-sized time boxes starting on different calendar dates, so our organization defined our own cadence of four-week cycles called *sessions*. The length was based on the amount of time we thought it'd take to run all the system tests, allow one round of defect fixes to be coded and integrated into the item under test, and verify the fixes. A four-week cycle gave a sliver a fighting chance to pass its tests and reach "Completed" in a single cycle.

The decision of which slivers would be delivered in any particular time box was negotiated with the technical leads of all subsystems. Most of the slivers included a list of "features needed from development before running this sliver", so it was possible to have reasonable (although long) conversations about what order would best accommodate all the different subsystem teams' needs.

We prepared our "session plans" showing which slivers would be tested in which timeboxes, and the technical leads compared the planned output of their time boxes with our sessions and slivers. Where our plan called for testing something that wasn't developed yet, either we moved the sliver later or the development team swapped their order around until we had a reasonable fit. Most teams didn't do a formal mapping between their chunks of development and our slivers, so this comparison was done on the fly.

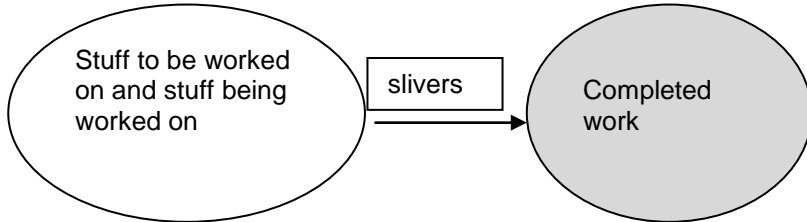
Once we had a final "session plan" which was agreed to by all partners, we were off to the races.

Monitoring Progress – the Cumulative Flow Diagram

Because of the enormous number of people and organizations involved, it's very easy for programs like these to get off track. A key aspect of Lean Product Development, and one that proved invaluable to us, is the emphasis on monitoring the performance of the *entire system* first, and examining individual pieces of the system only as needed. In our case, the entire system consists of the development teams **and** the integration & system test team. The intent is to put saleable features on the shelves for HP, and that takes all of us working together.

Lean Product Development tells you to draw a model of your system, showing activities and the flow between them. The model in turn tells you what to measure in order to monitor the overall progress. Typically one starts with a high-level, simple model.

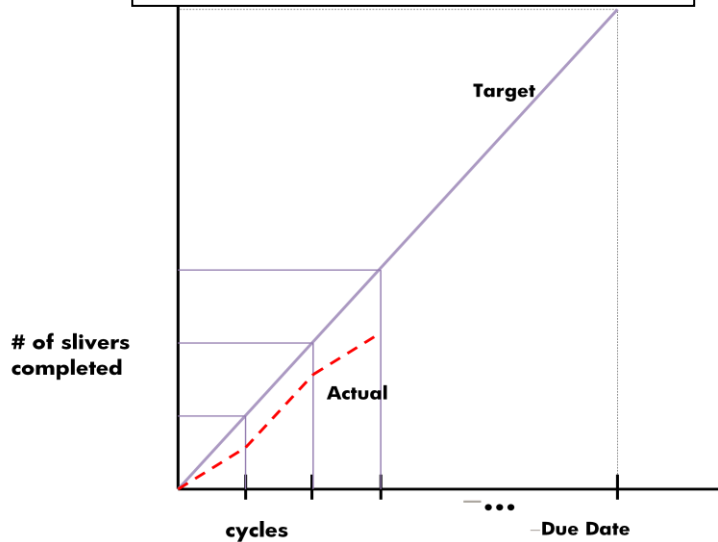
At the highest level, the system looks like this:



This simple model is monitored by a *cumulative flow diagram (CFD)*, which shows the actual number of slivers completed in each cycle.

Notice that we aren't showing how many *tests* are passing – we are showing how many *features* are passing testing. This is a more direct measure of the final, saleable output of the entire system.

A Cumulative Flow Diagram (CFD)

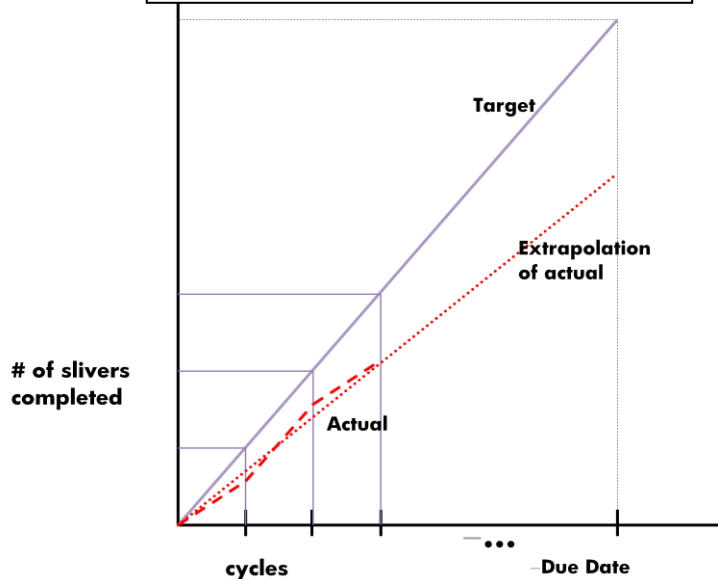


In this CFD, a simple extrapolation of the actual progress is possible after three or more cycles. This extrapolation clearly shows discrepancies between the plan and the predicted actual.

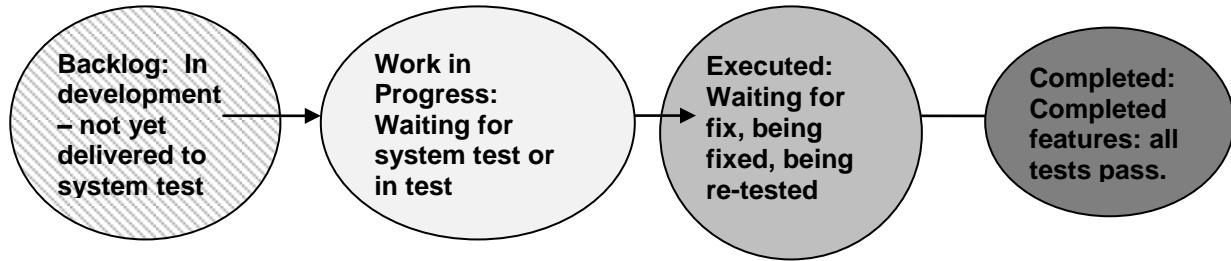
Any time fewer than the target number of slivers is delivered in a cycle, the program has fallen behind schedule. The following cycle will need to “catch up” by delivering more than the target in the next cycle.

So far, all the large programs where we used this method were behind schedule within the first few cycles. Some program teams found this difficult to accept at first and argued vociferously that the measurements must be incorrect, while others said “I thought we were behind, but now I know how much”.

CFD with extrapolation of actual

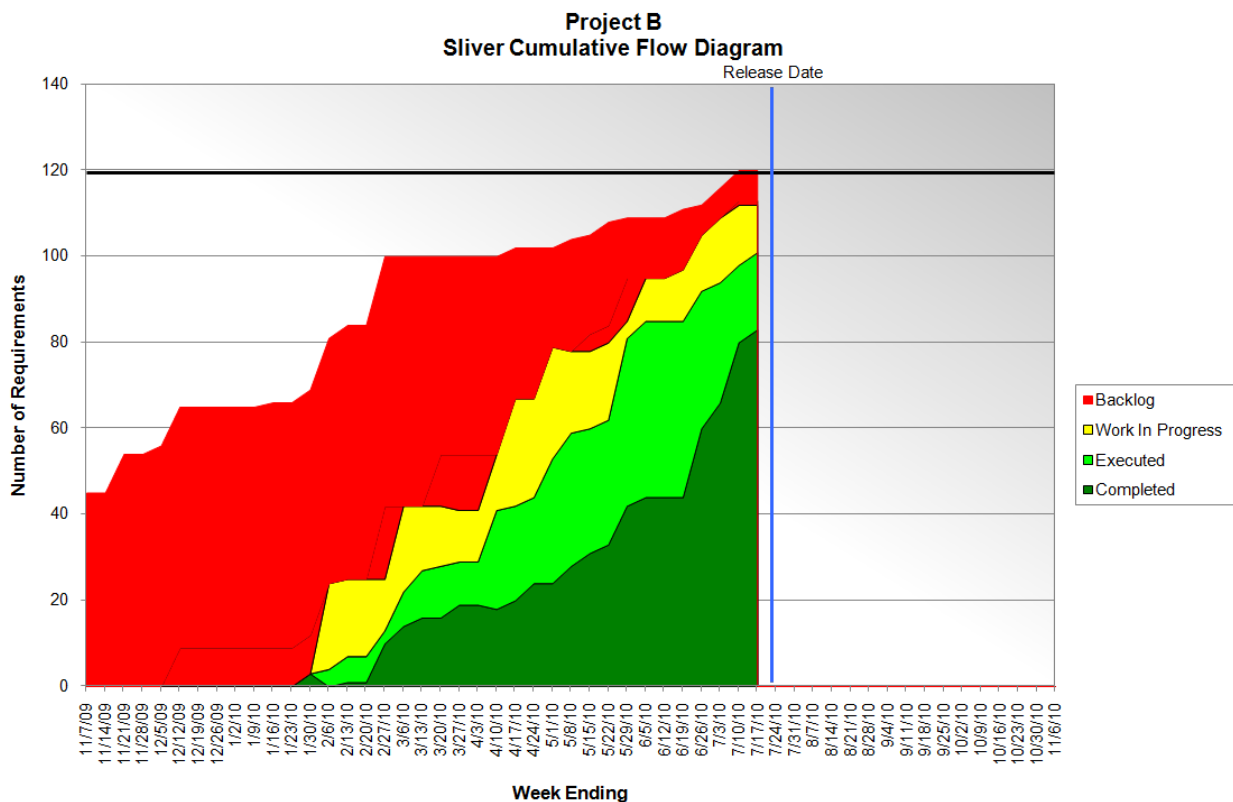


The first step in changing the throughput of any system is to understand why the throughput is what it is. This usually requires a more sophisticated model than the simple model on this page. The most frequently asked questions were around whether the problem was in System Test or in one of the Development organizations, so we broke up the first bubble in the previous model into three bubbles, showing the interplay between test and development.



Each sliver has a status flag, which can be set to any one of the statuses shown in the model below. The dates on which each sliver moves from state to state are tracked in QualityCenter. We wrote a program to extract the data from QualityCenter and graph it as a cumulative flow diagram.

This is a typical CFD.



With the slivers, their tests, and the CFD metrics program in place, we were ready to run a full program with Lean Product Development.

Project B runs under Lean Product Development

Project B is a high-end laser program similar to Project A. We started discussing the sliver-based system testing with this program about half-way through its development, well before system testing would usually start. The various organizations and the overarching program team were both willing to change their planned deliveries slightly to accommodate the sliver model.

As usual, the teams were eager to start system testing early. However, during the first 4-week session of system test, almost no slivers passed all their system tests.

A quick look at the defects found per test in the first session explained what was going on. The first session starts with an entrance or acceptance test suite, assessing the capability of the system to be system-tested. Nearly every test run in this suite discovered a defect, many of which were attributed to necessary code not having been delivered to system test.

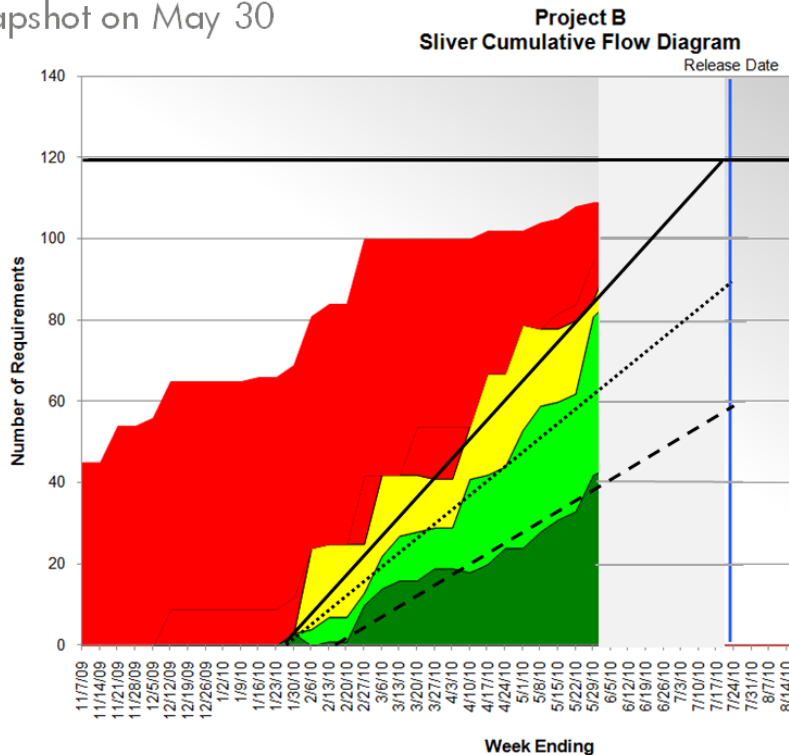
This may sound silly – unknowingly testing features which haven't been completed – but remember that these teams had been waiting until all features were complete before running these high-level system tests for any feature. Up to this time, there hadn't been a need to track exactly when all the work pertaining to a particular feature was completed across dozens of individual teams, so there wasn't a good high-level tracking system in place. Informal tracking didn't work well because the work could be spread across three separate organizations comprising hundreds of people.

Once the short entrance suite had been run, the test manager looked at these results and stopped the system testing. This forced the development organizations to discuss among themselves how they would know when a feature was ready to test, and they quickly improved their ability to track this. Stopping the testing also saved a good bit of money which would otherwise have been wasted on running tests which would repeatedly report that the same features weren't finished.

In the next cycle, a number of slivers were ready to test and the testing proceeded, finding and reporting useful defects to the various development teams. The bulk of the issues quickly changed from undelivered functionality to mostly defects. The test leads showed the CFD every week to the leads of the various development teams.

Things proceeded along for some weeks. Each week, the test leads reported which slivers weren't completed as planned, and what needed to be delivered from R&D in order to close these particular slivers. The CFD slowly showed a divergence between the plan and the extrapolation of the actual, indicating that the program was falling behind schedule.

Snapshot on May 30



On May 30, the sliver CFD showed:

- The rate of slivers delivered *into* system test (solid line) is a little higher than the rate of slivers *finishing* the first round of system test (dotted line).
→ This means that testing is not quite keeping up with development.
- The rate of slivers *finishing the first round of system test* (dotted line) is higher than the rate of slivers *entering the Completed state* (dashed line).
→ This means the fixing and re-testing isn't keeping up with either development or testing.
- The extrapolation of the Completion rate crosses the Target Release vertical line well below the target number of requirements
→ This means that the last sliver isn't going to complete until *well after* the target MR date.

Examining the open defects, it was obvious that the main problem was a large number of unfixed defects. The program manager was now monitoring the status of hundreds of defects, some of which were more than two months old. The program manager needed to track when they would be fixed, but also:

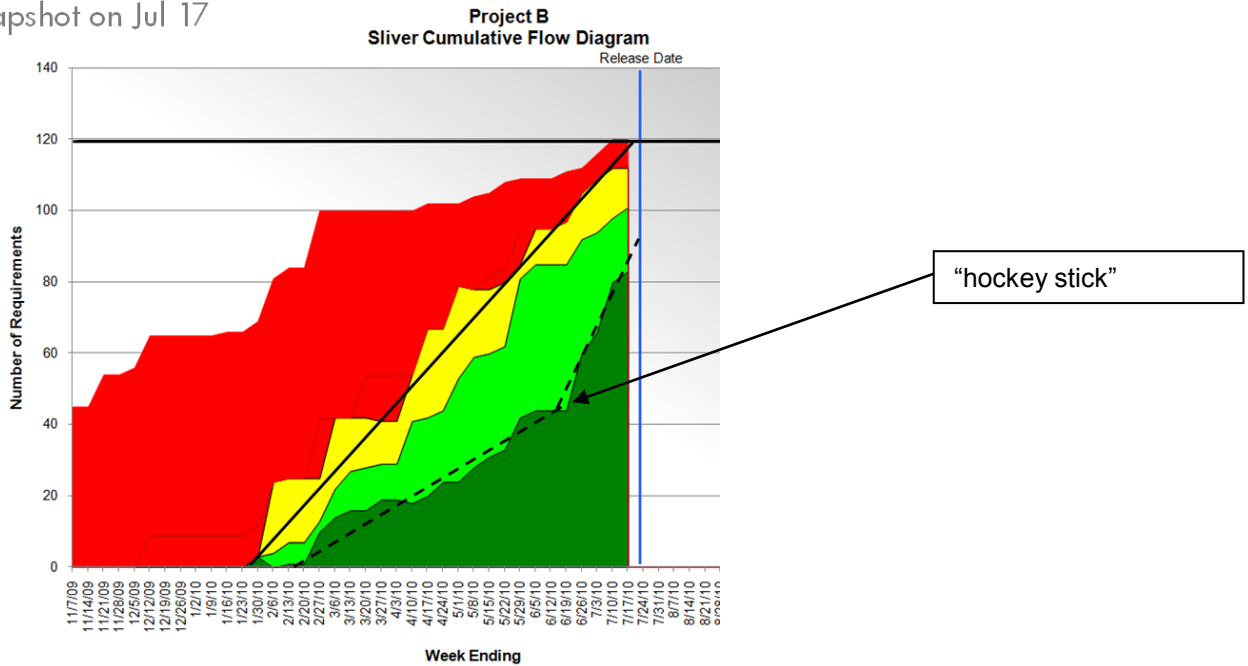
- What else is this defect blocking?
- What parts of the system are affected?

Even a couple dozen such defects is quite a lot to carry in your head. We found that the association of defects with slivers made it much easier for the program manager to understand and remember what was blocked and what was affected. Monitoring the status of slivers, rather than tests per se, also allowed the teams to target groups of defects which were blocking or affecting the same area of the system, and get the area sufficiently fixed to allow system testing to take place. This reduced the overall complexity of managing the project.

Of course, the reduction in complexity, while pleasant, didn't fix the divergence between planned and actual. The program was still behind schedule. The program team started to say "we need to see a hockey stick" in the CFD – that is, a sudden and sustained turn upwards. As usual, saying it didn't make it happen. After a couple of weeks of watching the growing divergence between the planned and actual, the program teams decided to take action. They stopped working on new features for a full week and

spent all their effort on fixing defects. The defect fix rate spiked sharply, but what we were really watching for was the hockey stick in the CFD – and sure enough, it appeared.

Snapshot on Jul 17



The large collection of defect fixes allowed a substantial number of slivers to re-enter testing and complete almost immediately. With the backlog mostly gone, defects found in subsequent slivers were fixed more quickly and the more rapid progress rate was sustained through the next months.

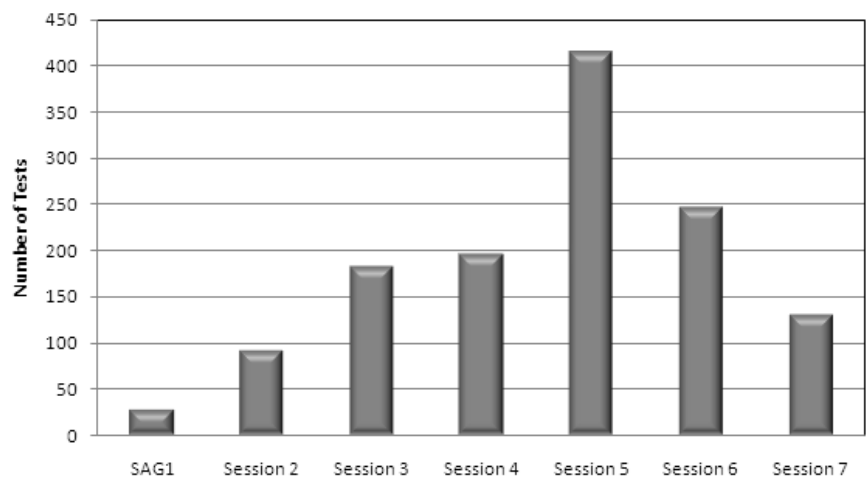
The End of Project B

As Project B nears its end, it is apparent that we didn't really achieve a smooth, linear delivery of features over the length of the project.

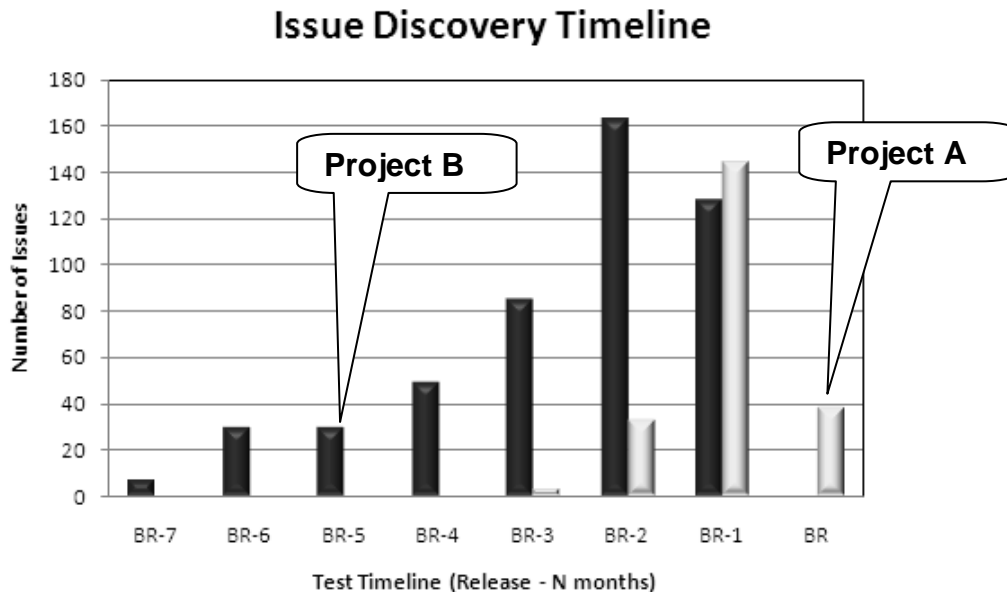
Tests Executed by Session in Project B

This chart shows tests executed by session in Project B.

Clearly the testing wasn't evenly spread throughout the seven months of system test.



However, we did see a very substantial change in feedback time between Project A and Project B. As seen in the defect discovery timeline, Project B started system test seven months before final release, compared to Project A which started three months before final release. This is definitely a substantial improvement in feedback time, since some developers received feedback three to five *months* (not weeks) earlier than they did in Project A.



Conclusion

We've really just begun our transition to Lean Product Development. The initial steps of the transition were quite costly, since we had to reorganize our system tests around features (as opposed to capabilities), *and* organize them in a way that would correspond to recognizable user stories, *and* organize them in a way that would not require enormous changes in the order in which work was done in each of the numerous subsystem teams. However, we got some immediate benefit due to the 25% reduction in the number of tests, so we had some time and resources to use on the transition.

Now that we have the foundation built and have started using Lean methods, we find that we can:

- Start system testing three to four months earlier than previously, thus cutting feedback time dramatically for half the defects.
- Show convincingly and credibly that a program is behind schedule and the improbability of catching up while still doing everything the same.
- Allow a program manager to understand quickly what features are blocked by which defects and thus simplify managing the program.
- Convince a program team to stop coding and fix for a week, despite the huge inertia inherent in large groups of teams.

However, we haven't yet started applying Lean to a program early enough to result in an efficient and well-understood order of development and integration across all the many subteams. We need to work out the order in which slivers will be delivered much sooner, near the very beginning of the program. This will provide a framework for the subsystem teams to align with each other and plan to deliver some features in full much earlier.

We're optimistic about achieving radical improvements in productivity next year as we continue to implement the Lean Product Development methods.

References

[Anderson 2004] Anderson, David J. 2004. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Pearson Education

[Anderson 2010] Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press; Sequim, WA, United States

[Jeffries 2001] Jeffries, Ron; Anderson, Ann; Hendrickson, Chet. 2001. *Extreme Programming Installed*. Addison-Wesley.

[Poppendieck 2007] Poppendieck, Mary; Poppendieck, Tom. 2007. *Implementing Lean Software Development: from Concept to Cash*. Addison-Wesley.

[Reinertsen 2009] Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas Publishing. Redondo Beach, CA, United States.

Lean Product Development Bibliography

Anderson, David J. 2004. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*. Pearson Education

Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press; Sequim, WA, United States

Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas Publishing. Redondo Beach, CA, United States.

Kniberg, Henrik and Skarin, Mattias. 2010. *Kanban and Scrum: Making the Most of Both*. C4Media.

Software Product Lines Bibliography

- Northrop, Linda and Clements, Paul. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- Software Engineering Institute at Carnegie Mellon: Software Product Lines
<http://www.sei.cmu.edu/productlines/>
- www.softwareproductlines.com

Note: Software Product Lines is sometimes called Software Product Families in Europe.

In Case You Wondered....

Lean Product Development isn't identical to Lean Manufacturing. They both use the same fundamental concepts from queuing theory, but manufacturing deals mainly with predictable tasks of similar sizes and relatively low variability, whereas product development deals with tasks which are inherently variable and dissimilar. Since the nature of the tasks is different, the application of queuing theory sometimes leads to different answers for Lean Manufacturing and Lean Product Development. For more on this subject, see Reinertsen 2009.