METHODS & TOOLS

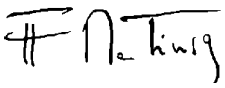
Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Fall 2007 (Volume 15 - number 3) www.methodsandtools.com

R.I.P.

Rest In Peace (from Latin requiescat in pace) is a sentence that typically appears on tombstones in Christian countries. Software development projects have usually the same destiny. Once they are finished, often the best thing that will happen is a little celebration for the project team. Nobody will formally look back at a project to understand what went well or wrong and why those things happen, so that this project can actually rest in peace and lessons learned could be used for the next projects. There is however an activity to analyse project after their completion. It is called "post mortem" analysis (so the RIP reference is not far) or "retrospective" in the agile approach. If you have not heard about this practice or never performed such activity, you should not be ashamed. I have never seen its implementation in my software development life. Even in the last Agile survey published by Version One, where a large majority of respondents are using an agile approach, only 39% of the participants are using this practice. Why?

The first explanation is that time is a rare resource in software development organisations. It is often associated to money in budget-driven companies. As there is a sense of urgency, people try to use their available time for the more essential (for them) activities. It is also difficult to tell as a project leader that you will spend some man/days after the project's completion to think about what went wrong. Are you not supposed to do things right in the first time? Our difficulty to create a context for constructive criticism is another issue faced when you try to institute some "post mortem" reflections. This is mainly due to the fact that it is difficult to separate the problems from the people that are associated to them. This is an important issue in organisations where professional excellence is a driver for monetary gains and career evolution. Will you speak openly about project problems if part of your salary is a bonus linked to achieving certain professional goals? The techniques proposed by Rachel Davies could help you to profit from lessons learned without creating personal issues, so that our past projects could really rest in peace and everybody could improve from a shared experience.



Inside

Measuring Integrated Progress on Agile Software Development Projects

Tamara Sulaiman, PMP, CST, tsulaiman @ solutionsiq.com,
SolutionsIQ, http://www.solutionsiq.com/
Hubert Smits, CST, hubert @ rallydev.com,
Rally Software, http://www.rallydev.com/

Summary

Earned Value Management (EVM) is a well known project management technique which measures the integration of technical performance, cost and schedule against planned performance within a given project. The result is a simple set of metrics providing early warnings of performance issues, allowing for timely and appropriate adjustments. In addition, EVM improves the definition of project scope, and provides valuable metrics for communicating progress to stakeholders. The information generated helps to keep the project team focused on making progress.

Despite the benefits that can be gained, EVM techniques they have been slow to be adopted in the private sector, especially in the area of software development. In many industrialized government sectors the acceptance rate is much higher. Many project managers in the private sector have hesitated to use EVM because they perceive the practices hard to implement.

Agile software development methods have been shown to be effective: software is developed faster, with higher quality, and better meeting changing business priorities and changing market conditions. The prevailing wisdom was that EVM techniques were too difficult to implement effectively on an Agile project, and that EVM could not easily cope with changing requirements. The EVM techniques have been adopted for Agile projects, and the correctness of the results has been proven [1]. AgileEVM is a light-weight, and easy to use adaptation of the traditional EVM techniques which provides the benefits of traditional EVM in Agile projects.

Among the benefits of AgileEVM is the ability to use the techniques on simple Agile projects (single team, short duration) as well as on scaled Agile projects (multiple teams at the program or portfolio level). Earned Value and Planned Value calculations for individual teams can be combined for a concise overall picture of the performance of the program against the plan. Because the AgileEVM metrics are expressed the same way that traditional EVM metrics are expressed, a ‘roll-up’ across hybrid programs (mix of traditional and Agile teams) is also possible.

Why Use Earned Value Management Techniques for Software Development?

It has traditionally been seen as difficult to accurately forecast final project costs for any software development project. The actual performance of a project (teams ahead or behind schedule or budget) cannot be converted easily to a cost forecast. The best option that project managers have is to use notoriously inaccurate estimates of cost of work for this purpose.

In a 1998 article, Fleming and Koppelman [2] assert that “The single most important benefit of employing earned value is the cost efficiency readings it provides…”. Earned Value Management has been a recognized project management technique since the 1960’s. It is the subject of in-depth study by the Project Management Institute’s (PMI) College of Performance Management and is now included as a standard in the “Guide to the Project Management Body of Knowledge” published periodically by the PMI. EVM integrates the areas of technical performance, schedule and actual cost to provide metrics for work actually accomplished. By

comparing the earned value (EV) with the planned value (PV) the actual progress on the project is compared against the expected progress which yields valuable information. Using this information, metrics assessing cost efficiency and schedule efficiency are calculated. Metrics forecasting the expected cost to complete a project and total expected cost of a project based on past project performance are derived.

What is Agile and Scrum?

Agile (or lightweight) Software Development Methods have been developing over the last 15 years. They are a response to the often document-centric and heavy processes that are generally used for software development. The core principles of the Agile methods are:

- Iterative and Incremental: software is not developed in a multi-month effort, but in a series of short (2 – 4 week) iterations;
- People centric: the whole team is responsible for planning, designing and delivering the increments. Teams are small, 7 to 10 people and scaling happens by adding teams to the project. Teams are encouraged to be self-organizing;
- Enabling change: Agile methods allow for requirement changes at the end of every iteration, enabling a better tuning of product requirements with the delivery process;
- Business focus: in every iteration, only the features that are of the highest importance for the business are delivered;
- Regular inspection of product and process: at the end of every iteration the delivered product features are demonstrated, and the effect on the product is considered. Also at the end of every iteration the team inspects the way they delivered the features and agrees on process improvements.

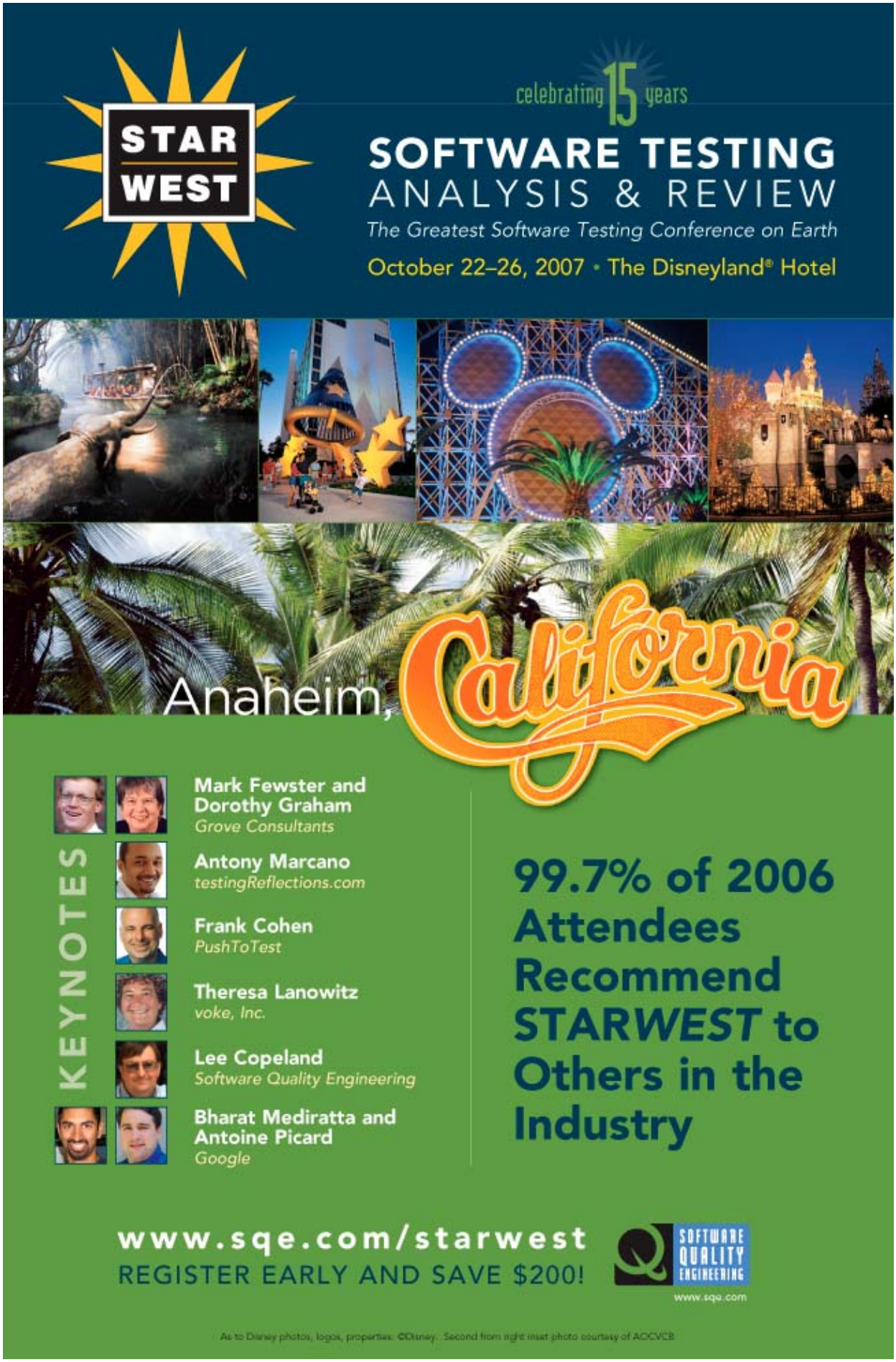
A full description of Agile Methods is beyond the scope of this article, we suggest books by Peter Schuh (Integrating Agile Development in the Real World) and Craig Larman (Agile and Iterative Development: A Manager's Guide) as good introductions to the topic.

What do all those new words mean?

This document is heavy on jargon, some often used terms are:

- Agile Methods – The collection of lightweight software development methods that have been developed based on the Agile Manifesto. Examples are Extreme Programming (XP), Scrum and Feature Driven Development.
- Product Owner – The person responsible for the success of the product in the market, and therefore entitled to prioritize the needed features of the product.
- Delivery Team – The group of people responsible for the delivery of the artifacts that together make up the product. They are responsible for delivering the right quality and can therefore determine and estimate the tasks involved in the delivery of the product features.
- Product backlog – a prioritized list of features that make up the product as desired by the product owner.
- Iteration or Sprint – a one to four week period in which the delivery team produces working (accepted) product features.
- Feature – a product component that the product owner and customer recognize and value.

Advertisement – StarWest Conference - Click on ad to reach advertiser web site



What is AgileEVM?

AgileEVM is an adapted implementation of EVM that uses the Scrum framework artifacts as inputs, uses traditional EVM calculations, and is expressed in traditional EVM metrics. AgileEVM requires a minimal set of input parameters: the actual cost of a project, an estimated product backlog, a release plan that provides information on the number of iterations in the release and the assumed velocity. All estimates can be in hours, story-points, team days or any other consistent estimate of size. The critical factor is that it must be a numerical estimate of some kind. In this article we will use story-points as a measure of story size and velocity.

What does AgileEVM provide that the burndown chart doesn't?

Agile methods do not define how to manage and track costs to evaluate expected Return on Investment information. Therefore the iteration burn-down and burn-up charts (as used in Scrum) do not provide at-a-glance project cost information. Agile metrics neither provide estimates of cost at completion of the release nor cost metrics to support the business when they consider making decisions like changing requirements in a release. AgileEVM does provide this information, and is therefore a excellent extension to the information provided by burn-down charts.

What is our baseline?

We are using three datapoints to establish the initial baseline:

- The number of planned iterations in a release;
- The total number of planned story points in a release;
- The planned budget for the release.

In order to calculate the AgileEVM metrics, there are four measurements needed:

- The total storypoints completed;
- The number of Iterations completed;
- The total Actual Cost;
- The total story points added to or removed from the release plan.

How to calculate Planned Value, Earned Value and Actual Cost?

The comparison of the Earned Value (EV) against the Planned Value (PV) lies at the core of the EVM. Planned Value is the value of the work planned for a certain date. It is the entire budget for work to be completed at the planned date. In Scrum terminology: it is the sum of the estimated feature sizes for all the features up until the planned date.

Earned Value is the value of work completed at the same date as used for PV. Earned Value is not synonymous with actual cost, nor does the term refer to business value. Earned Value refers to technical performance (work) "earned" against the baseline or work planned. In Scrum terminology: it is the sum of the estimated story points for the features up until the calculation date.

Actual Cost is what the name implies: the cost in dollars to complete a set of features.

The reader will be aware by now that the terminology used in this article is specific to EVM. Our day-to-day language has sometimes different interpretations for terms like 'Earned Value'. The article will continue to use the correct EVM vocabulary, as this is important for the use of these metrics in an audience that is familiar with the technique.

An example to clarify these three concepts. With a total project budget of $ 175,000, and having completed one out of four Iterations, we have this product backlog and these actuals:

Feature	**Estimate (storypoints)**	**Completed (storypoints)**	**Actual Cost (1000s dollars)**
Welcome Screen	10	10	15
Advert - Splash Screen	20	20	30
Login Screen	10	10	20
Personalized Google Ads	20		
Catalog Browser	20		
Catalog Editor	10		
Shopping Basket Browser	5		
Shopping Card Editor	25		
Check-out Process	20		
Invoice Calculation	10		
Credit Card Verification	10		
PayPal Payment Handling	20		
Order Confirmation Email	20		
Totals	**200**	**40**	**65**

With this information (which should all readily be available in any Scrum project) it is easy to calculate our three base values:

Expected Percent Complete equates to the number of completed iterations divided by the number of planned iterations (in our example, after Iteration 1 we should be at 25% complete);

- Planned Value for a given iteration is the Expected Percent Complete multiplied by the Total Budget (25% of $ 175,000 = $ 43,750);
- Actual Percent Complete equates to the total number of storypoints completed divided by the total number of storypoints planned (40/200 = 20% complete);
- Earned Value is calculated by multiplying Actual Percent Complete by the Total Budget (20% of $ 175,000 = $ 35,000.

AgileEVM works by comparing the current release plan (taking changes in requirements into account) against actual work performed. We can see at a glance that our project is in trouble: the Earned Value is less than the Planned Value (EV = $ 35,000 and PV = $ 43,750).

It is important to note that in AgileEVM there is no credit for partial completion. The backlog items are either done or not done (0 or 100%.) In keeping with Agile terminology: a backlog item is only 'complete' and story points awarded when the customer accepts the item as 'done'.

Analyzing the AgileEVM metrics –
using Cost Performance Index and Schedule Performance Index

The Cost Performance Index (CPI) gives a measure of efficiency. It shows how efficiently you are actually spending your budget dollars compared to how efficiently you planned to spend them. It is calculated by dividing Earned Value by the Actual Cost. In our example, CPI is 35,000 / 65,000 = .53.

A CPI of 1 indicates that you are spending your budget to accomplish work at the rate that you had planned to spend it. A CPI less than 1 means that you are over budget i.e. you are spending your budget less efficiently than planned because your Earned Value is larger than your Actual Cost.

CPI > 1	CPI = 1	CPI < 1
Under Budget	On Budget	Over Budget
EV > AC	EV = AC	EV < AC

The Estimate at Complete is the forecast of the total amount that we will need to spend to complete the planned work, based on the actual work accomplished. The easiest calculation is to divide the Total Budget by the Cost Performance Index. While this is not the most accurate calculation, it is accurate enough to identify trends in time to take corrective action. Our Estimate at Complete is $ 175,000 / .53 = $ 330,188 we predict to be 47% over budget.

Advertisement – PairCoaching.net

PairCoaching.net is a group of individuals that met in the agile community.
After delivering our workshops at several international events, we considered it time to offer these same workshops in an open format.

We deliver pragmatic, people oriented best practices for software development teams. In each workshop we focus on one aspect of teamwork. Our participants go home with knowledge and experience they can apply immediately in their day-to-day work. **Each workshop is led by 2 trainers.** Our courses are also available on demand on-site in EMEA.

Are you a blogger? Find out how to apply for the 10% Blog discount code.

Agile Kick Start Belgium 2007/09/26 by Vera Peeters and Yves Hanoulle

Retrospectives for Agile Teams London 2007/11/21 by Rachel Davies and Emmanuel Gaillot

Team 2.0 McCarthy BootCamp Belgium from 2007/11/04 to 2007/11/09 by Jim & Michele Mccarthy This is the first time the McCarthy's bring their teamwork laboratory to Europe. We believe BootCamp is the most cost-effective tool available to get any group of people working together as a high-performance team. Don't find enough developers? Why not optimize your current team instead?

Ruby & Ruby On Rails Belgium from 2007/11/29 to 2007/12/04 by Pascal Van Cauwenberghe and Hans Vandenbogaerde

Leadership Day Belgium 2007/12/14 by Ignace Hanoulle and Yves Hanoulle

www.paircoaching.net

The Scheduled Performance Index (SPI) compares Earned Value with Planned Value and is calculated by dividing the Planned Value into the Earned Value (our SPI is 35,000 / 43,750 = .8 – we are 20% behind schedule). The analysis of the SPI is comparable to the CPI analysis:

SPI > 1	SPI = 1	SPI < 1
Ahead of Schedule	On Schedule	Behind Schedule
EV > PV	EC = PV	EV < PV

And an Estimated Completion can be estimated by dividing the SPI into the Planned Iterations, in our example we planned to finish the work in 4 Iterations, and expect to need: 4 / .8 = 5 Iterations.

Agile allows me to change the backlog after each iteration, does Agile EVM cope?

Yes, the AgileEVM method does take into account the changes to the backlog that may be introduced after each iteration. By using the changes to the backlog (the added or removed estimated storypoints caused by added or removed features), you effectively 're-baseline' after every iteration. The effect of re-computing the AgileEVM after every iteration, is a validation of the modified backlog against release plan. You are validating that you will be able to complete all of the work planned for the release within both schedule and budget. This gives the Product Owner (who 'owns' the backlog) early information about the effects of his changes, early enough to reconsider changes if the affect the release plan negatively.

How often should Earned Value be measured?

Iteration boundaries are well suited to calculate the AgileEVM, it can be done more often, which is applicable for a release that contains only a few Iterations. EVM has been proven to be statistically accurate as early as when 15% of the planned features are completed, which makes EVM and AgileEVM an important metric to report to your stakeholders. The important consideration is that you have the correct cumulative values available at the boundaries that you choose to calculate your AgileEVM.

How can EVM be 'rolled up' across project teams in a program?

When teams in a program are comparable in size and velocity then the AgileEVM calculations can be made over the whole program. Often teams are not comparable, they work on different parts of the project, with different velocities, different storypoint values and different project costs.

When the latter scenario occurs (teams have different characteristics), calculate the Earned Value for each team, and add them up to a Total Earned Value. The example below shows how we did this for a program with 3 teams. Also calculate the Planned Value for each team and add them up for a Total Planned Value. From there on all of the EVM calculations remain the same, with the caveat that the Actual Cost needs to be the actual cost across teams to calculate a total CPI across a program. The CPI and Estimate at Complete are useful indicators of how the program is doing overall, but does not give you an indication of any particular team that may be in trouble.

Team	Total Budget	Planned Value	Earned Value	Actual Cost	CPI	SPI	Estimate at Complete
Team A	$ 300k	$ 150k	$ 150k	$ 150k	1	1	$ 300k
Team B	$1,000k	$ 575k	$ 500k	$ 625k	.8	.86	$ 1,250k
Team C	$ 800k	$ 175k	$ 200k	$ 180k	1.11	1.14	$ 720k
Program Totals	**$ 2,100k**	**$ 900k**	**$ 850k**	**$ 955k**	**.89**	**.94**	**$ 2,360k**

In this example, the total program is forecast to be over budget by 11% (1 minus CPI), or $260,000, and is 6% over time. By looking at the results for each team, you can see that Team A is currently performing according to planned cost and schedule. Team B is 20% over planned budget, and 14% behind planned schedule. Team C is performing better than planned, 11% under budget and 14% ahead of schedule.

Conclusion

Using these simple performance metrics in conjunction with the more typical Agile metrics (the Iteration burn-down and release burn-up charts for example) provides objective analysis to share with teams, management and customers. The early warning that the AgileEVM metrics show, validates changes to release plans and provides business with the opportunity to make priority trade-off decisions early in the project lifecycle.

References

1. Agile EVM Research Paper; http://www.solutionsiq.com/agile_index.html
2. Earned Value Project Management, A Powerful Tool for Software Projects; http://www.stsc.hill.af.mil/crosstalk/1998/07/value.asp

Advertisement – Software Quality Assurance Zone - Click on ad to reach advertiser web site

The Software Quality Assurance Zone is a repository for resources concerning software testing (unit testing, functional testing, regression testing, load testing), code review and inspection, bug and defect tracking, continuous integration. The content consists of articles, news, press releases, quotations, books reviews and links to articles, Web sites, tools, blogs, conferences and other elements concerning software quality assurance. Feel free to contribute with your own articles, links or press releases.

Learn and contribute to the software quality assurance zone

www.sqazone.net

Lean Configuration Management
Evolving the CM Discipline Through the Agile Paradigm Shift

Jens Norin
http://intellijens.se

Configuration management, as a discipline for supporting software development, has been around for half a century and has evolved into standard practice within traditional software development processes. One of the key purposes of configuration management is to *control changes* made to the software product.

Lean software principles focus on delivering customer value in a steady flow and eliminating unnecessary process waste. One way to implement lean is to introduce agile software development methods, such as Scrum or XP. Agile values also have a lot of focus on how to handle changes, but unlike traditional CM, agile methods are being more *adaptive to changes*.

The increasing popularity of agile development methods is putting new demands on the traditional CM discipline. A working CM environment is essential for the rapid nature of agile development methods, but the CM process and the CM role has to be adapted to the present evolution of software development methods as well as automated tools.

This article discusses lean principles and agile values within a CM scope and also introduces a method to classify the CM discipline in relation to development method and level of tool automation, and finally shares some of the experiences of the author.

Traditional Configuration Management Definition

Nearly every publication on Configuration Management (CM) has its own definition of the subject. I have chosen to quote a few examples to illustrate their common properties and to be able to compare those further on with lean and agile thoughts.

Especially note how they all mention how to deal with changes, where traditional CM handles changes by controlling them. Agile ideas also focus a lot around how to handle changes, but talks more on how to be adaptive to changes. They are targeting the same basic problem, but with slightly different approaches.

IEEE - *"Configuration Management is the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items."* [8]

CMM - *"…Software Configuration Management involves identifying the configuration of the software (selected software works products and their descriptions) at given points in time, systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the software lifecycle. The work products placed under software configuration management include the software products that are delivered to the customer (for example the software requirements document and the code) and the items that are identified with or required to create these software products (for example the compiler)…"*. [3]

RUP - *"The task of defining and maintaining configurations and versions of artifacts. This includes baselining, version control, status control, and storage control of the artifacts."* [21]

Here is a brief list of practices that usually are included in the definition of CM:

- Identify configuration items
- Version control of configuration items
- Release management
- Build management
- Controlling changes
- Tracking status
- Auditing

Lean principles applied on the CM process

CM in itself is a supporting discipline not delivering any direct customer value. But what if we turn it around a bit and look at the CM process as the deliverable? Being careful not to sub-optimize, we could ask ourselves who is the customer and end user of the CM process? And what is value to them?

Advertisement – MKS Integrity - Click on ad to reach advertiser web site

I would say the development team is the primary customer and user of the CM process. There are still more stakeholders that benefit the CM discipline, like project managers, product management etc and indirectly the end user/customer. Value, primarily to the development team, but also the other stakeholders, must be a visible and streamlined CM process that allows them to develop features fast in order to fulfill their main task, which is to deliver value to their customer, the end user. This means that CM should support and help the project to deliver customer value more rapidly.

Below is a brief discussion of Poppendiecks' seven principles of lean software development, from a CM perspective. [17]

Eliminate Waste – this is what Lean is all about, identifying and eliminating unnecessary waste. When it comes to CM a typical source for waste is the CCB (Change Control Board) or the change control process. CCB-meetings are often held periodically and a Change Request is often visited over several meetings updating the status in between, making the turnaround time unnecessarily long.

Running the project according to Scrum makes the traditional CCB/change process obsolete. New features, bug fixes and change requests should be prioritized together by the product owner. By shortening the iteration length the cycle time for change requests can be shortened. If we need to shorten it even further we can set aside a certain amount of time or resources in an iteration to focus only on incoming change requests (still prioritized). For example one person in the team is focusing on change requests during the whole iteration and the next iteration someone else is doing it, in a rolling schedule.

Another common source of waste related to CM is code ownership. The strong code ownership seen in many large organizations has a key responsible person for a code module. This key responsible person is the only one allowed to do changes in this module. When implementing a feature many modules can be affected, which could lead to a situation where we are queuing work at the module responsible person, and queues are waste. We need to move from a module driven development process towards a feature driven. [15]

Build Quality In – This is about building quality into the process/product instead of adding it in a test phase at the end. Traditionally it is often the configuration manger's task to keep track of the defect list, or queue. A lean approach would be to *prevent* defects by finding and fixing them before they end up as defects in a test phase. This can be done by automating the test process having automated tests of different levels for unit- acceptance- and regression testing and eventually also Test Driven Development (TDD).

Building quality into the CM process is also about enabling continuous improvement by implementing agile retrospectives at the end of every iteration, and making sure CM issues are brought up.

Create Knowledge – Feedback is knowledge, and to get knowledge through feedback we have to identify the feedback loops and run them frequently. This means that the CM infrastructure must allow frequent releases, with an efficient build process and frequent integrations between parallel activities. An efficient agile build process utilizes continuous integration and automatic testing enabling immediate feedback to the developer, as well as frequent deliveries, and feedback, to the customer.

Another CM aspect of creating knowledge is to spread the daily operating CM routines throughout the team. The team should have the knowledge to perform these daily operations themselves without having to rely on an explicit CM role. If this knowledge is missing it could be introduced by periodically pairing an inexperienced team member with an experienced CM [14]. The CM role still exists within the organization, but with a more enterprise focus towards CM strategies, tool support and facilitating agile development for projects and teams.

Defer Commitment – In short this means that we should not make decisions until we have to. More strictly it says that for irreversible decisions we should first decide when the last responsible moment for taking a decision takes place. When that last responsible moment occurs we have the most information to make our decision.

For reversible decisions we can just go ahead and make a decision and simply change it if it turns out to be a bad decision, instead of just sitting around and not making any decision at all.

Putting a CM aspect on this would rather be to create a flexible CM environment allowing reversible decisions. Creating a branching structure that allows parallel development is one example.

Deliver Fast – For some reason it has been widespread in the software industry that if we do things slow and careful we get higher quality, especially when it comes to requirements gathering. However if we instead deliver fast and frequent we get rid of waste, we get valuable feedback and we are able to find hidden errors much earlier, all leading to higher quality.

We can make the CM process support fast deliveries and higher quality by introducing continuous integration and automated testing into the build process together with frequent incremental deliveries.

Respect People – If you give respect you will get respect and only by showing respect and listening to people you will benefit their full creativity, ideas, loyalty and engagement. In today's tough competition you cannot afford not to. Agile leaders respect the workers and support and facilitate their work. In the same way an agile CM process is supporting and facilitating the work of the team.

Optimize the Whole – It is not enough to only optimize the CM process by introducing Lean CM. That would be to sub-optimize. A lean organization optimizes the whole value stream, from the receiving of a customer request until the deployment and customer satisfaction. This article tells how to adapt a more traditional CM process to a lean and agile environment.

Agile methodology and CM

Agile CM is maybe a more widespread expression than Lean CM. However I like to think that agile methodologies and engineering practices can be seen as ways to implement LEAN software principles, and what I actually aim for is to make the CM discipline leaner focusing on flow and minimizing waste.

Most agile methods do not mention any explicit CM routines or practices, simply because it is out of the context of the agile methods. For example XP and Scrum are more of project frameworks that do not explicitly include CM in their scope. XP does however rely on working CM routines for some of its engineering practices. [12][19]

Agile CM is not limited to CM for agile projects, but is when you actually apply agile values on the CM discipline [2]. The agile values are captured in the Agile Manifesto [11]. If we analyze the agile manifesto within a CM context we get the following discussion:

Individuals and interactions over processes and tools – The CM discipline is traditionally heavy on tools. Even though it should be obvious that tools should support the process, it is far too common that the process is adapted to the tools. We should not forget that software is developed by humans and the process should support appropriate human behavior and not obstruct it. A more pragmatic approach is preferred over a dogmatic.

Working software over comprehensive documentation – Since CM is a supporting discipline we can make agile CM support agile development by shifting focus towards optimizing the "value chain" instead of creating waste such as unnecessary CM documentation. The production of necessary CM documentation should be automated where possible.

Customer collaboration over contract negotiation – I have seen many CCB routines that have ended up in endless contract negotiations to decide whether an issue is a change request (customer cost) or a trouble report (project cost). In these cases the change process is often constructed in a way that aims to prevent the customer from filing changes. In agile methods, such as Scrum, the product backlog replaces the need for a CCB. Instead the backlog is a continuously prioritized list containing all that is planned to be done, including change requests and trouble reports. The resources should constrain what is finally implemented, not the original contract.

Responding to change over following a plan – Traditional CM seeks to control change, while agile values adapt and respond to change. The common property between CM and agile methods is that they should facilitate change and not prevent it.

CM performance

The performance of CM in a project or an organization can be measured with two parameters: the degree of CM process formality and the degree of CM (tool) automation. Plotting the two parameters in a diagram gives the following picture. Here we classify each parameter into relatively high and relatively low measures according to our perception of current practices.

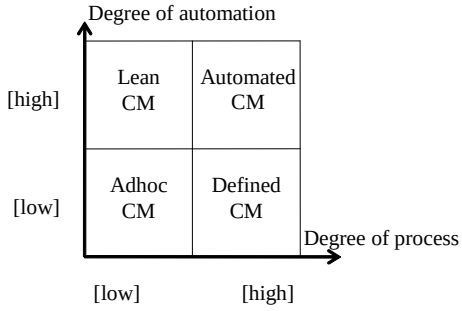


Figure 1: CM performance quadrant.

Ad hoc CM = low degree of defined process and low degree of automation
Defined CM = high degree of defined process and low degree of automation
Automated CM = high degree of defined process and high degree of automation
Lean CM = low degree of defined process and high degree of automation

Case Studies

The following ideas are based on real world experiences from several different projects in different organizations. I choose to describe two of these organizations, which have had the largest impact, in more detail.

Company A – This is a multinational retail company where an effective global supply chain is a key success factor. IT is a supporting discipline of great importance for fulfilling their supply chain. Company A has adopted a customized RUP adaptation as the central development method for the last few years. Most projects within the organization have interpreted this in a way that most closely resembles a waterfall approach. This makes it a very heavy and defined process with a heavy CM overhead and no tool automation.

Most projects are running the RUP adaptation and have the prescribed dedicated CM role within the project. The same CM resource is often shared between several projects. The company also uses a gate model to manage their projects and productivity is in general perceived as quite low. However, also mentioned in coming sections, there are isolated agile initiatives outside of the general enterprise process.

Company B – This is a multinational company producing consumer electronic products in a highly competitive business. IT is a central part of their products, which are very hi-tech, containing advanced custom hardware and software, with short life-cycles. They have a development method that closely resembles waterfall method and a gate model for managing products. They are forced to be productive because of the extremely tough competition in their market. Given the circumstances they succeed quite well, but this is often perceived to be on the staff's expense. However there is plenty of room for improvements, as for example the lead times for change requests are very long.

The CM role in company B is split up into two roles. One which mostly resembles a change control manager on a product level, who is occupied with change requests, CCB-issues and release management. The other part of the CM role is heavily biased towards integration and building.

Ad hoc CM

Ad hoc CM is here defined as when the degree of process is low and the degree of tool automation is low. The typical setup would be having a simple versioning tool, but no dedicated CM role and no formal CM routines. This is sufficient for the smallest of projects and predictable environments.

The degree of process, however, could also be more highly developed, but still be slim and low in waste. This is better, but the lack of automation creates unnecessary manual overhead.

Ad hoc CM in its most extreme form implies no process and no tools, which could also be referred to as "chaos CM" or "no CM". This might be enough for projects with very small teams, maybe only one team member. But in this case there is no traceability, no version handling tool, no support for change management and should generally be avoided because of the potential risk exposure and unexpected high costs these can lead to.

The CM process has for many organizations developed from an initial low degree of process together with a low degree of automation. From there the CM process evolves to a defined CM process concurrently with the evolution of the rest of the software development method, maybe triggered by an initiative to introduce RUP, waterfall, CMM or any other traditional method.

The relation with company A began several years ago, and started with projects categorized as having an Ad Hoc CM process. Between five to ten years ago this seemed to be the standard setup at company A for small and medium sized projects. At the time, company A had no central development process and no central function promoting development methods. It was instead up to the individual projects to decide, often ending up with no particularly formal process. This led to a diverse environment and maintenance problems. However the productivity was interestingly enough very high. There are still isolated projects that for some reason have little or no CM process, but in the ones witnessed this have only led to bad traceability and general confusion.

Defined CM

The defined CM process is defined as when there is a relatively high degree of defined process, but a low degree of tool automation. This could typically be the case when an organization has introduced a quality framework such as CMM or a development method such as RUP. In these methods we have a dedicated CM role with a heavier process including CCB routines.

Company A has the last 5 years been centralizing many of their software systems and functions. As a step in this process they also standardized the development method throughout the organization by introducing RUP as the common development method. The CM role is pretty much taken from off the shelf RUP. This has shifted focus from producing code towards distinct handovers, rigid planning and very formal change management. Unfortunately this has led to a serious decrease in productivity within the organization.

Company B, with its very large development organization, is also considered to be put into this category. The code is very modularized and they have a culture with strong code ownership. This, together with the large number of differently configured products in varying states of development makes CM a challenging task to handle. The development organization is also large enough for them to require many CMs/integrators in the project in some form of hierarchy that mirrors the architecture of the product. The problem they experience with this setup is that they have very long lead times for change requests.

Automated CM

We define Automated CM as the rather uncommon situation of having a relatively high degree of CM process while having a relatively high level of tool automation. Our experience says that organizations that benefit from automated tool support have also learned to benefit from agile methods.

At company A, however, there are isolated projects that are moving towards more CM tool automation. This is due to agile supporters among the developers that are introducing continuous integration into a RUP project. This might shift attitudes within the project towards an interest in agile development, but does not do enough for productivity, since the development process is effectively blocking a large part of the possible benefits gained from the automation.

Lean CM

Lean CM shares the same values as agile development methods and lean principles. Lean CM supports the empirical nature of software development and is more adaptive to changes instead of being predictive. To support agile development processes, tools use must be automated.

These prerequisites to Lean CM disqualifies traditional development methods like the waterfall or RUP, as these describe a very much more controlled and heavy CM process based on predictive planning with high amounts of waste.
Company A have lately started to introduce Scrum into a few projects with very successful results. They have in general experienced an increase in productivity. They have managed to do this with the above recipe of not having a dedicated CM, and in coexistence with a legacy RUP environment.

In company A there is also at least one project that has been completely excepted from the RUP policy for very long. This project is doing XP and is able to deploy into production every 3 weeks. This project is also perceived as being very productive.

Creating Value with Lean CM

By applying lean software principles to the CM discipline we can enable projects or organizations to create more value and reduce waste. If we look at this from an economical perspective we are actually improving *productivity*. There are three basic ways to do this [18]:

- Reduce software development effort.
- Streamline development processes.
- Increase customer value.

Reducing software development effort

This is basically about reducing development costs by eliminating investments in features that are not valuable to the customer. Lean principles support this by reducing waste, especially by avoiding overproduction. Agile methods encourage an empirical approach to development as opposed to the waterfall approach where detailed and complete requirements are done upfront. This is achieved by identifying and shortening feedback loops and also to manage requirements in a prioritized product backlog.

Streamline development processes

More mature industries (outside software industry) have been increasing productivity for decades by streamlining processes and making more efficient use of resources, a good example is the car industry with Toyota as a classical example [9]. A mature software development organization is one that can rapidly and repeatedly transform customer requirements into deployed high quality code. This is where lean CM really comes into play by supporting the agile methods and lean principles in order to reduce waste, as well as using tools to further support the process and automate tasks.

Increase customer value

The key issue here is to understand what represents value to the customer, and often the customers do not know this themselves, especially in the beginning of the project. We must try

to understand how the customer will be using the product to be able to deliver value. Since CM is a supporting discipline it doesn't have a great impact on customer value. However CM can help sweep the path for the team by enabling short feedback loops.

Conclusion

We have seen that traditional CM handles changes by controlling them while agile methods handle changes by responding and adapting to them. Can these two disciplines really co-exist, or does one make the other obsolete? Well actually CM is more important than ever for enabling an agile environment. The short feedback loops and some of the agile engineering practices rely heavily on working CM processes. But a different mindset could be used, where CM is seen as supporting and enhancing the agile methods.

My top three measures to be taken to transfer the organization towards lean CM, a bit provocative to some, would be:

- **Cancel the traditional CCB** – Instead introduce the backlogs of Scrum where new features and changes can be prioritized together. Badly setup CCB routines just add waste and long lead times to the change process, because important change decisions are held up just waiting for the CCB to convene, This introduces wasted lead time if the decision affects the critical path of the project.
- **Remove the dedicated project CM** – Instead let the daily operative CM routines be handled by the self-organizing team, while the CM experts in the organization focus more on enterprise issues such as CM strategy, tool support, and supporting the teams when needed.
- **Automate tools** – Automate the build environment by introducing continuous integration, automatic testing etc. using tools such as CruiseControl. [5]

References

1. Alistair Cockburn, *Agile Software Development*, Addison Wesley, 2002.
2. Brad Appleton, Robert Cowham, Steve Berczuk, *Defining Agile SCM: Past, Present & Future*, CMCrossroads.com, 2007
3. Carnegie Mellon Univ. Software Engineering Inst. *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison Wesley, 1995.
4. Craig Larman, *Agile & Iterative Development, A Manager's Guide*, Addison Wesley, 2004.
5. *CruiseControl*, http://cruisecontrol.sourceforge.net/
6. David E. Bellagio and Tom J. Milligan, *Software Configuration Management Strategies and IBM Rational ClearCase*, IBM Press, 2005.
7. Goldratt, E., *Critical Chain*, North River Press, 1997.
8. IEEE-Standard-729-1983, *Standard Glossary for Software Engineering, ANSI/IEEE.*
9. Jeffrey Liker, *The Toyota Way*, McGraw-Hill, 2003.
10. Jens Norin, Daniel Karlström, *Lean Configuration Management – Supporting Increased Value Delivery from Agile Software Teams,* 6th Conference on Software Engineering Research & Practice in Sweden (SERPS'06), 2006
11. Kent Beck et al, *Manifesto for Agile Software Development*, http://www.agilemanifesto.org, 2001.

12. Kent Beck, *eXtreme Programming - Embracing Change*, Addison Wesley, 1999.
13. Kent Beck, *Test Driven Development: By Example*, Addison Wesley, 2002.
14. Laurie Williams, Robert R. Kessler, Ward Cunningham and Ron Jeffries, *Strengthening the Case for Pair-Programming*, IEEE Software (Vol. 17, No. 4), 2000.
15. Martin Fowler, *Code Ownership*, http://www.martinfowler.com/bliki/CodeOwnership.html, 2006.
16. Martin Fowler, *The New Methodology*, http://www.martinfowler.com/articles/newMethodology.html, 2005
17. Mary and Tom Poppendieck, *Implementing Lean Software Development: From Concept to Cash*, Addison Wesley, 2006
18. Mary Poppendieck, *Agile Contracts: How to Develop Contracts that Support Agile Software Development*, Agile 2005 Conference, 2005.
19. Mike Beedle and Ken Schwaber, *Agile Software Development with SCRUM*, Prentice Hall, 2001.
20. Peter H. Feiler, "Configuration Management Models in Commercial Environments", CMU/SEI-91-TR-7, Carnegie-Mellon University/Software Engineering Institute, March 1991.
21. Philippe Kruchten, *The Rational Unified Process: An Introduction*, Addison Wesley, 2003.
22. Poppendieck LLC, "Software Development Productivity", http://www.poppendieck.com/publications.htm
23. Stapleton, J. *Dynamic Systems Development Method: The Method in Practice*, Addison Wesley, 1997.
24. The SEI Software Capability Maturity Model

Refactoring Your Development Process with Retrospectives

Rachel Davies, Agile Experience Limited.
http://www.agilexp.com

Software development is not a solitary pursuit, it requires collaboration with other team members and departments. Most organisations establish a software lifecycle that lays down how these interactions are supposed to happen. The reality is that many teams find that development process does not fit their needs or is simply not followed consistently. It's easy to grumble when this happens and it can be frustrating if you have ideas for improvements that don't get taken up. This article offers a tool that may help your team get to grips with process improvements. This tool is the Retrospective. This article aims to explain what you need to do to facilitate a Retrospective for your team.

Background

A Retrospective is meeting which gets the whole team involved in the review of past events and brainstorming ideas for working more effectively going forward. Actions for applying lessons learned are developed for the team by the team.

The term "Retrospective" was coined by Norman Kerth author of "*Project Retrospectives: a handbook for team reviews*" [1]. His book describes how to facilitate three-day off-site meetings at the end of a project to mine lessons learned. Such retrospectives are a type of post-implementation review – sometimes called *post-mortems*! It seems a pity to wait until the end of a project to start uncovering lessons learned. In 2001, Extreme Programming teams adapted retrospectives to fit within an iterative development cycle [2]. Retrospectives also got added to another agile method, Scrum [3] and nowadays it's the norm for teams applying agile development to hold many short "heartbeat" retrospectives during the life of the project so that they can gather and apply lessons learned about their development process **during** the project rather than waiting until the end.

Learning from Experience

Experience without reflection on that experience is just data. Taking a step back to reflect on our experience is how we learn and make changes in our daily lives. Take a simple example, if I hit heavy delays driving to work then it sets me thinking about alternative routes and even other means of getting to work. After some experimentation, I settle into a new routine.

No one wants to be doomed to repeating the same actions when they are not really working (some definition of madness here). Although a retrospective starts with looking back over events, the reason for doing this is to change the way we will act in the future – retrospectives are about creating change not navel-gazing. Sometimes we need to rethink our approach rather than trying to speed up our existing process.

Retrospectives also improve team communication. There's an old adage "a problem shared is a problem halved". Retelling our experiences to friends and colleagues is something we all do as part of everyday life. Often on a project, no one person knows the full story of events. The whole story can only be understood by collating individual experiences. By exploring how the same events were perceived from different perspectives, the team can come to understand each other better and adjust to the needs of the people in their team.

Defusing the Time-bomb

Let's move onto how to run an effective retrospective. Where a team has been under pressure or faced serious difficulties tempers may be running high and relationships on the team may have gone sour. Expecting magic to happen just by virtue of bringing the team together in a room to discuss recent events is unrealistic. Like any productive meeting, a retrospective needs a clear agenda and a facilitator to keep the meeting running smoothly. Without these in place, conversations are likely to be full of criticism and attributing blame. Simply getting people into a room to vent their frustrations is unlikely to resolve any problems and may even exacerbate them.

Retrospectives use a specific structure designed to defuse disagreements and to shift the focus to learning from the experience. The basic technique is use structured group activities to slow the conversation down – to properly explore different perspectives on events before drawing conclusions.

The Prime Directive

By reviewing past events without judging what happened, it becomes easier to move into asking what could we do better next time? The key is to adopt a systems thinking perspective. To help maintain the assumption that problems arise from forces created within the system rather than destructive individuals, Norm Kerth declared a Prime Directive for retrospectives that he proposed is a fundamental ground-rule for all retrospectives.

> Prime Directive: *Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given what was known at the time, his or her skills and abilities, the resources available, and the situation at hand.*

This purpose of this prime directive is often misunderstood. Clearly, there are times when people messed up – maybe they don't know any better or maybe they really are lazy or bloody-minded. However, in a retrospective the focus is solely on making improvements and we use this Prime Directive to help keep us in constructive mode. Poor performance by individuals is best dealt with managers or HR department and the Prime Directive firmly sets such conversations outside the scope of retrospectives.

Getting started with Ground Rules

To run an effective retrospective someone needs to facilitate the meeting. It's the facilitator's job to create an atmosphere in which team members feel comfortable talking.

Setting ground-rules and a goal for the retrospective helps it to run smoothly. There are some obvious ground-rules that would apply to most productive meetings – for example, setting mobile phones to silent mode. So what additional ground-rules would we need to add for a retrospective? It's important for everyone to be heard so an important ground-rule is "No interruptions". If in the heat of the moment this rule is flouted then you can try use a "talking stick" so only one person is talking at a time – the person holding the talking stick token (the token does not have to be a stick – it could be an object from your office such as a mug - teams I have worked with have used a fluffy toy which is easier to throw across a room than a mug).

Advertisement – Agile Development Practices Conference - Click on ad to reach advertiser web site

Once the ground-rules for the meeting are established then they should be written up on flipchart paper and posted on the wall where everyone can see them. If people start to forget the ground-rules then it is the facilitator's job to remind everyone. For example, if someone answers a phone call in the meeting room then gently usher them out so that their conversation does not disrupt the retrospective.

Safety Check

Another important ground rule is that participation in all activities during a retrospective is optional. Some people can feel uncomfortable or exposed in group discussions and it's important not to exacerbate this if you want them to contribute at all. When a team do their first few retrospectives, it's a useful to run a "Safety Check" to get a sense of who feels comfortable talking. To do this run an anonymous ballot, ask each person to indicate how likely they are to talk in the retrospective by writing a number on slips of paper using a scale 1 to 5 (where 1 indicates "No way will I share my point of view" and 5 "Completely comfortable to talk openly") – the facilitator collects these slips of paper in, tallies the votes and posts them on a flipchart in the room. The purpose of doing this is for the participants to recognise that there are different confidence levels in the room and for the facilitator to assess what format to use for subsequent discussions. Where confidence of individuals is low, it can be effective to ask people to work in small groups and to include more activities where people can post written comments anonymously.

Action Replay

Sportsmen use the action replay to analyse their actions and look for performance improvements. The equivalent in retrospectives is the Timeline.

Start by creating a space (on the wall) for the team to post events in sequence that happened during the period they are reflecting over; moving from left to right – from past to present. Each team member adds to the timeline using coloured sticky notes (or index cards). The facilitator establishes a key for the coloured cards. For example, pink – negative event, yellow – neutral event and green – positive event. The use of colour helps to show patterns in the series of events. This part of the meeting usually goes quickly as team members work in parallel to build a shared picture.

The activity of creating a timeline serves several purposes – helping the team to remember what happened, providing an opportunity for everyone on the team to post items for discussion and trying to base conversations on actual events rather than general hunches. The timeline of event is a transient artefact that helps to remind the team what happened but it is not normally kept as an output of the retrospective.

Identifying Lessons Learned

Once a shared view of events has been built, the team can start delving for lessons-learned. The team is invited to walk the timeline from beginning to end with the purpose of identifying; "What worked well that we want to remember?", "What to do differently next time?" and "What still puzzles us?".

The facilitator reads each note on the timeline and invites comments from the team. The team works to identify lessons learned both good and bad. It's important to remind the team at this stage that the idea is to identify areas to focus on rather than specific solutions as that comes in Action Planning.

As a facilitator, try to scribe up a summary of the conversation on a flipchart (or other visible space) and try to check with the team that what you have written accurately represents the point being made. Writing down points as they are originally expressed helps show that a person's concerns have been listened to.

In my experience, developers are prone to talking at an abstract level – making general claims that are unsubstantiated. Such as, "the architect is never around", "we always have problems with environments", and "there are too many meetings". As a facilitator, it's important to dig deeper and check assumptions and inferences by asking for specific examples to support the claims being made.

Action Planning

Typically, more issues are identified than can be acted on immediately. The team will need to prioritise issues raised before starting action planning. The team needs to be realistic rather than wishful thinking mode. For an end of iteration retrospective, no more than three and five actions would be a sensible limit.

Before setting any new actions, the team should review whether there are outstanding actions from their previous retrospective. If so then it's worth exploring why and whether the action needs to be recast. Sometimes people are too ambitious in framing an action and need to decrease the scope to something they can realistically achieve. For each action, try to separate out the long-term goal from the next step (which may be a baby-step). For example, a long term goal might be "implement continuous integration" and a short term goals might be "create a script to automate the build", "refactor tests to remove dependencies on live database so tests run in less than 10 minutes", "install Bamboo tool".

The team may even decide to test the water by setting up a process improvement as an experiment where the team take on a new way of working and then review its effectiveness at the next retrospective. Also it's important to differentiate between short-term fixes and attempting to address the root cause. Teams may need both types of action – a book, which provides a nice model for differentiating between types of action, is Edward De Bono's "Six Action Shoes"[4].

Each action needs an owner responsible for delivery plus it can be a good idea to identify a buddy to work with that person to make sure the action gets completed before next retrospective. Some actions may be outside the direct sphere of influence of the team and require support from management – to get that support the team may need to sell the problem! Your first action in this case, is to gather evidence that will help the team convince their boss action is required.

Wrapping-up

Before closing the retrospective, the facilitator needs to be clear what will happen to the outputs of the meeting. The team can display the actions as wallpaper in the team's work area. Or the team may choose to use a digital camera to record notes from flipcharts/whiteboards so the photos can be upload a shared file space or transcribed onto a team wiki. Before making outputs visible to the wider organisation the facilitator needs to check with the team that they are comfortable with doing this.

Perfecting Retrospectives

To run a retrospective it helps to hone your facilitation skills - a retrospective needs preparation and follow through. The facilitator should work through the timings in advance and vary the activities used every now and again. For example, rather than using a timeline for data-gathering you might ask the team to focus on a specific aspect of their process (such as design reviews or testing) and use a different activity to explore this. A good source of new activities is the book "Agile Retrospectives" by Esther Derby & Diana Larsen. A rough guide to timings is a team need 30 minutes retrospective time per week under review so using this formula allow 2 hours for a monthly retrospective and a whole day for a retrospective of a several months work.

In addition, to planning the timings and format, the facilitator also needs to review: Who should come? Where to hold the meeting? When to hold the meeting? When a team first starts with retrospectives they will find that they come up with plenty of actions that are internal to the team. Once the team has it's own house in order then they usually turn to interactions with other teams and it's worth expanding the invitation list to include people from outside the immediate team who can bring a wider perspective.

As a team lead or manager, it can sometimes be hard to play the role of neutral facilitator when you have opinions and observations you would like to share as a participant. If you work alongside other teams that use retrospectives then it may be possible to take turns to facilitate them for each other.

As standard practice at the end of my retrospectives, I gather a Return On Time Invested (ROTI) rating from participants and if you are trying to build a team of facilitators in an organisation then gathering such participant feedback is likely to be useful to help new facilitators get a sense of how they are doing and can be used to gauge the effectiveness of any type of meeting.

Finding a suitable meeting space can make a big difference. It may help to pick a meeting room away from your normal work area so that it's harder for people to get dragged back to work partway through the retrospective. Where possible try to avoid boardroom layout – sitting around a large table immediately places a big barrier between team members – and instead look for somewhere that you can setup a semi-circle of chairs. You also need to check the room has at least a couple of metres of clear wall space or whiteboards. I have learned that when an offsite location is booked for a retrospective it's important to check that there will be space to stick paper up on the wall. I have sometimes been booked to facilitate retrospectives in fancy boardrooms with flock wallpaper, bookcases and antique paintings and had to resort to using doors, windows and up-ended tables to create temporary wall space.

As for timing, when working on an iterative planning cycle, you need to hold the retrospective before planning the next efforts. However, running retrospective and planning as back-to-back meetings will be exhausting for everyone so try to separate them out either side of lunch or even on separate days.

Final Words

I am sometimes asked, by people wanting to understand more about retrospectives, "Can you tell me a story that demonstrates a powerful outcome that resulted from a retrospective?". I have come to realize that this question is similar to "Can you tell me about a disease that was cured by taking regular exercise?".

I have worked with teams where running regular heartbeat retrospectives made a big difference in the long term but because the changes were gradual and slow they don't make great headlines. For example, one team I worked with had an issue of how to handle operational change requests that came in during their planned product development iterations. It took us a few months before we established a scheme that worked for everyone but without retrospectives it might have taken a lot longer.

The power of regular retrospectives and regular exercise is that they prevent big problems from happening so there should be no war stories or miraculous transformations! Embracing retrospectives helps a team fine-tune their process at a sustainable pace.

References

1. Project Retrospectives: A Handbook for Team Reviews by Norman L. Kerth. Dorset House. ISBN: 0-932633-44-7
2. “Adaptation: XP Style” XP2001 conference paper by Chris Collins & Roy Miller, RoleModel Software
3. Agile Project Management with Scrum by Ken Schwaber. Microsoft Press, 2004. ISBN: 978-0735619937
4. Six Action Shoes by Edward De Bono HarperCollins 1993. ISBN: 978-0006379546
5. Agile Retrospectives: Making Good Teams Great by Esther Derby and Diana Larsen. Pragmatic Programmers 2006. ISBN: 0-9776166-4-9

Advertisement – Software Development Tools Directory - Click on ad to reach advertiser web site

Are you looking for the right development tools for your task? A new directory of tools related to software development has been launched. It covers all software development activities: programming (java, .net, php, xml, c/c++, ajax, etc), testing, configuration management, databases, project management, modeling, etc.

Search and reference software development tools on www.softdevtools.com

Mass Customizing Solutions

Jack Greenfield, Microsoft
http://blogs.msdn.com/jackgr/

Software Factories are a new paradigm, described in an award winning and best selling book [1], for bridging the gap between broad market platform technologies and custom solutions for business process automation in the enterprise. Software Factories promise to bring efficient mass customization to the enterprise application market by promoting the formation of supply chains used to provision software solutions.

The Customer Dilemma

Historically, there has been a large gap between the general purpose, horizontally oriented, broad market technologies supplied by software vendors and the specialized, vertically oriented, custom solutions developed to automate proprietary business processes. The gap is created by a mismatch between the diversity of customer requirements and economies of scale in the software industry. On the one hand, no two customers are identical. Each has unique requirements that can only be completely satisfied by a custom solution. On the other hand, software vendors must target large homogeneous markets in order to keep development costs within reason, essentially treating their customers as if they are identical. The gap is both a serious problem and a significant growth opportunity for the software industry.

Current Solution Strategies

Enterprises have tried to close the gap using a combination of third party products, especially packaged applications, and custom development, often performed by external service providers. Neither of these solution strategies fully satisfies most customers.

Packaged Applications

Packaged application vendors like SAP, Oracle and Microsoft have successfully delivered custom solutions from predefined assets, including requirements, logical and technical architectures, implementation components, test suites, deployment topologies, operational facilities, maintenance plans and migration pathways, on a larger scale than had previously been seen in product based solutions in the enterprise market.

Unfortunately, packaged applications are not a panacea. While they can generally be customized to a great extent to address the needs of individual customers, the levels of customization they admit do not let the typical customer fully follow their preferred proprietary business practices. In order to use a packaged application, the customer must generally adopt business practices mandated by the software. This forced conformity makes it hard for the customer to achieve competitive advantage over other companies running the same applications.

Another problem with packaged applications is that they are typically monolithic, meaning that they attempt to span the entire gap in a single product based solution using closed architectures that are fully accessible only to the application vendor. These characteristics of the software make it hard to integrate with products from other vendors, forcing the customer to depend on the packaged application provider to meet most or all of its business process automation needs, or to invest in clumsy integration strategies that often prove to be hard to use, slow, resource intensive, insecure, and hard to maintain as the packaged applications, the platform technologies, and the customer’s business needs change over time.

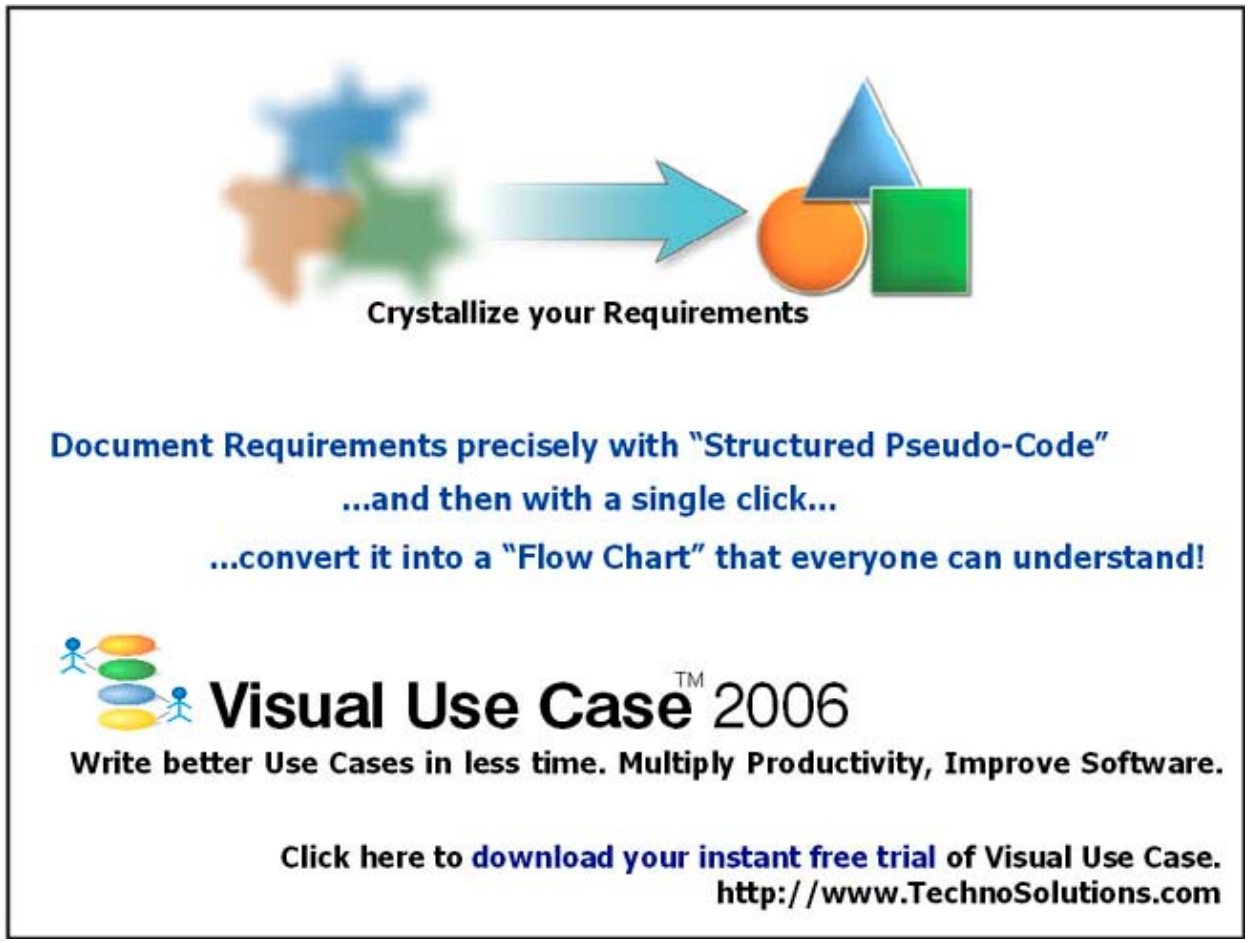
Enterprise customers are looking for a more flexible approach that better satisfies their requirements, enabling them to achieve competitive advantage through differentiation. They are also looking for more open and more modular architectures that will give them more options in the marketplace, and that will preserve their business information and their investments in business process automation across changes in underlying products and technologies.

Custom Solution Development

With custom solution development, software is built to order, usually by third party service providers working closely with the customer's internal information technology organization. When carried out successfully, custom solution development produces software that fully satisfies the customer's unique and proprietary requirements, enabling the customer to differentiate itself from competitors in the hope of achieving competitive advantage.

Unfortunately, custom solution development is often expensive, time consuming and error prone, and rarely delivers all of the features originally planned. The resulting software is generally of lower quality than commercially developed software products, in terms of usability, reliability, performance and security, and is often hard to operate, and hard to support and evolve as business requirements and technologies change.

Advertisement – Visual Use Case - Click on ad to reach advertiser web site



Enterprise customers have explored many strategies for solving these problems, such as using widely marketed but marginally effective methodologies based on low fidelity general purpose modeling languages, and more recently using agile development methods and off shoring. While these strategies can yield some marginal gains, they cannot offer systemic improvements.

Most enterprises have tried multiple methodologies, and many are still looking for one that delivers consistently. Widely marketed methodologies are generally either too thin and too weak to help the customer manage the challenges of developing software in the real world, or too heavy and too formal to accommodate the unpredictable nature of real world projects. Most give generic abstract advice rather than concrete guidance for a specific type of deliverable. You get the same advice whether you're building eBay or embedded software for an anti-lock braking system, although these domains obviously have very different requirements.

Most of the models used by these methodologies serve only as documentation, and then usually only in the early stages of projects, either because the modeling tools don't generate production quality code, or because. they don't stay synchronized with the code, and therefore cannot capture key design decisions. They are often thrown away when the code starts taking shape.

A Different Approach

Software Factories represent a different approach to bridging the gap between platform technologies and custom solutions. They combine the best characteristics of the packaged application and custom solution development strategies without the problems described above, enabling rapid, predictable and inexpensive delivery of highly customized product based solutions that meet the unique requirements of individual customers. They achieve these results by changing the way individual suppliers build solution components, and by promoting the formation of supply chains that distribute cost and risk across networks of suppliers who collaborate to produce a wide range of custom solutions tailored for a large number of individual consumers from a core set of products.

A Different Kind Of Methodology

Like other methodologies, Software Factories help developers follow known good practices and apply known good patterns. Unlike other methodologies, however, they recognize that different domains require different solution strategies. Consequently, instead of a one-size-fits-all approach, the methodology calls for the development of many factories, each providing guidance for a specific domain. Also, unlike other methodologies, which offer guidance from ivory tower experts, Software Factories assume that the people best qualified to provide the guidance are the practitioners who work in the target domain. For this reason, there are two parts to the methodology, one part for factory authors and another part for factory users.

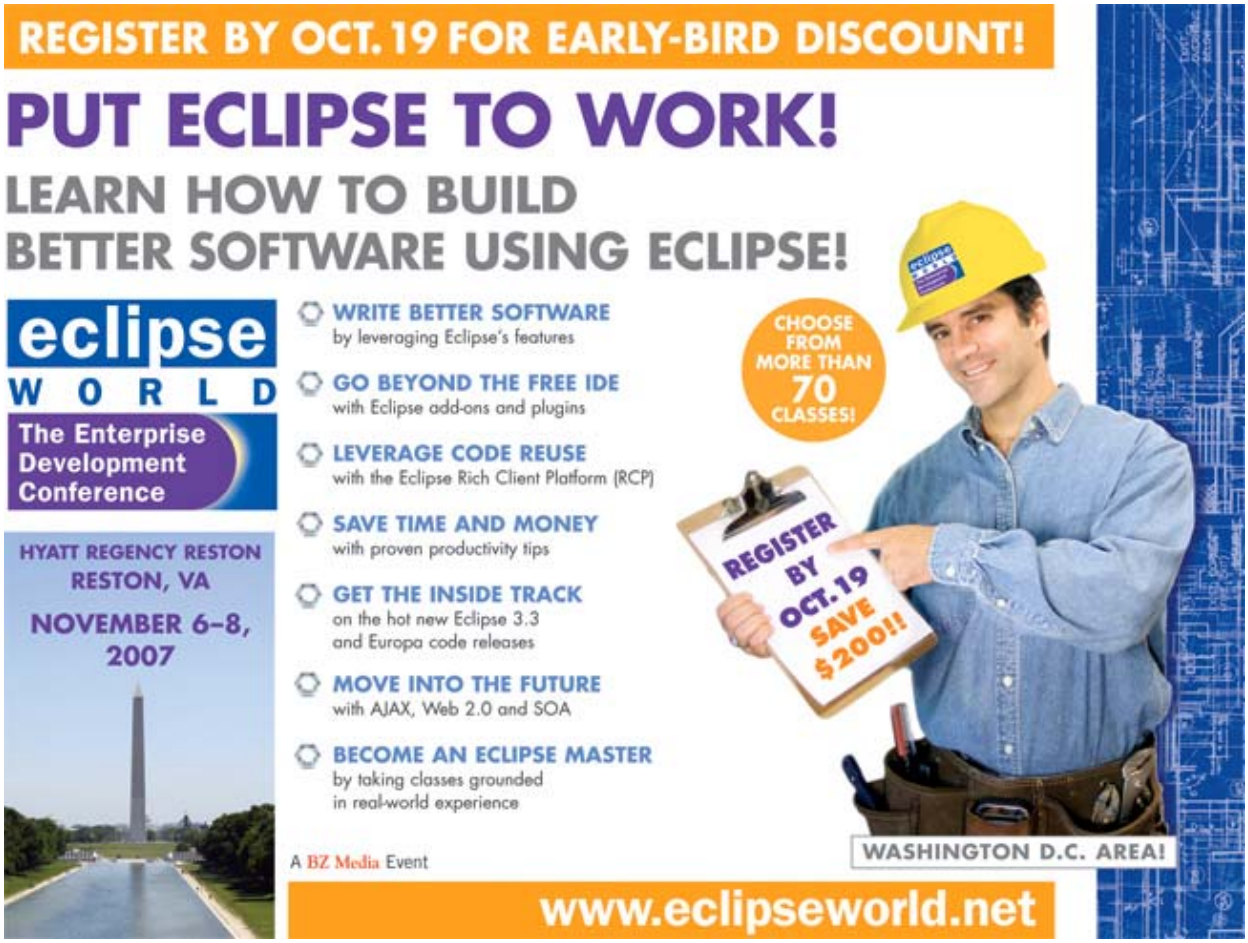
In order to support the definition and differentiation of individual factories, the methodology starts by identifying and classifying architectural styles for frequently encountered families of solutions, such as web portals, smart clients and connected systems. It then captures information about each solution family, such as the artifacts needed to build its members, the technologies used and their placement within the architecture, required technology configurations, and best practices across the life cycle, including key decisions points and trade-offs. Factories containing only this type of information are called architectural factories. More specialized factories can also be created targeting common functions, such self service portals and ecommerce applications. These are called functional factories. Still more specialized industry factories can be created targeting specific industries, segments or solution areas, such as factories for banking self service portals, or online music stores.

Like other methodologies, Software factories describe processes that span part or all of the solution life cycle. Unlike other methodologies, however, Software Factories supply reusable assets to help users enact the processes. Some of these assets are documents, such as patterns, guidelines and check lists, some are tools, such as designers, wizards, and scripts, some are assets used by tools, such as templates, models and configuration files, and some are executables, such as libraries, frameworks and sample code.

The methodology is based on a familiar and time tested pattern for improving productivity and predictability. Known good patterns and practices in the target domain are harvested, refined and encoded into domain specific tools and runtimes. The tools are then used to rapidly select, adapt, configure, complete, assemble and generate solutions that use the runtimes. Familiar examples include graphical user interface builders and frameworks, web site development kits and platforms, and database management tools and servers.

Until recently, applying this pattern has been technically and economically viable only for broad market, horizontally oriented domains served by vendors offering general purpose tools and runtimes. Software factories change the market dynamics by making the pattern viable for much smaller, more vertically oriented domains, enabling vendors closer to the customer to supply special purpose tools and runtimes to much narrower market segments.

Advertisement – EclipseWorld Conference - Click on ad to reach advertiser web site

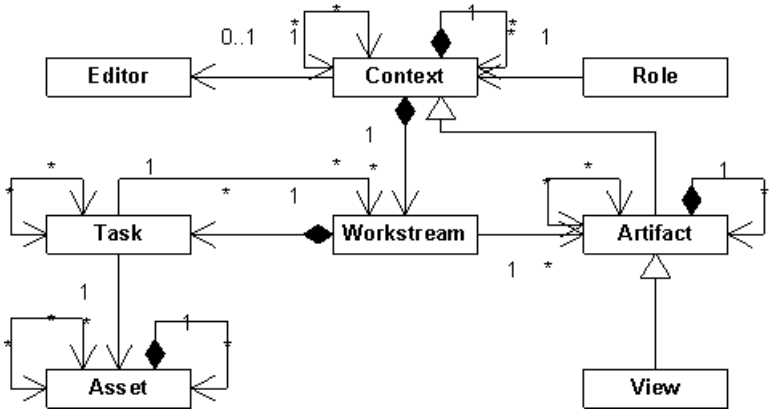


What Is A Software Factory?

A software factory is a specialized development and runtime environment that supplies an integrated set of special purpose assets encapsulating proven patterns and practices, including tools, processes and content. Examples of content assets include partial or prototypical life cycle artifacts, such as requirements, logical and technical architectures, test suites, deployment topologies, operational facilities, maintenance plans and migration pathways, and implementation artifacts, such as guidelines, patterns, code samples, templates, libraries, frameworks, models, and configuration files. These assets are used by solution builders to accelerate recurring tasks, such as process partitioning, contract definition, feature configuration, deployment validation, requirements tracing, and impact analysis, for a specific family of products or solutions.

The assets are delivered in a structured and installable package called a software factory template. The assets are usually customizable, and the organization of the template is designed to make it easy to select, adapt, configure, complete, assemble and parameterize the assets, enabling the factory to produce a wide range of solutions with varying features and operational qualities.

At the core of a factory is a description of the factory and its contents called a schema. Like a database schema, a software factory schema is a model that can be interpreted by people and tools. At the core of the model is an architecture framework for the target product family. (Architecture frameworks are described below.) It describes the assets that comprise the factory, how they are packaged, and how they are installed, customized and applied. The structure of the factory schema is illustrated by the following diagram.



How Do Software Factories Work?

Software Factories integrate four technologies to create a whole that is greater than the sum of its parts: software product lines, model driven development, guidance automation and architecture frameworks.

Software Product Lines

A software product line is a set of systems sharing a set of managed features that satisfy the specific needs of a particular market segment, developed from a common set of core assets in a prescribed way [2]. In a product line, new solutions are developed by assembling partial solutions and/or by configuring generic ones. Only the unique features of each solution must be specified because the common ones can be safely assumed.

In a software product line, some variations in requirements map predictably to variations in artifacts and processes. The decision to support personalization in a web service portal, for example, maps predictably to the addition of tables and columns in the database schema, to changes in web pages and page flow, to changes in business logic, to service contracts offered by components, and to new tasks or changes in tasks in development, testing, deployment, operations, maintenance and migration.

Software product line engineering practices are reported to reduce custom development by 40% to 80% for the typical solution. Lower gains are too small to economically support product line development and operation, while higher gains are nearly impossible to realize consistently in practice. The portions of the solution not covered by the product line are custom developed, often using a Request for Price Quotation (RPQ).

The asset base used in a product line is usually reverse engineered from existing systems or subsystems that are partially or wholly prototypical of the family of systems targeted by the product line. The assets are then adapted as necessary when new family members are developed, to address variations not yet anticipated by the product line. The revised assets are then customized and applied to produce the family member.

Guidance Automation

The goal of guidance automation is to help practitioners know what needs to be done at any given point in the life cycle of a given type of solution and to provide reusable assets, such as help topics, code samples, templates, wizards, workflows, libraries, frameworks and tools, that help them do what needs to be done. It is implemented using installable packages containing organized sets of integrated assets supporting commonly occurring use cases. The tool and content assets are contextualized by attaching them to relevant tasks in the processes and to relevant parts of the solution architecture. The tasks can then be scoped using pre and post conditions, so that the project workflow may vary subject to the conditions. This allows tasks to be activated or to come into focus only when relevant to the state of the solution, and to be deactivated or to go out of focus only when their exit criteria have been satisfied.

Model Driven Development

The goal of model driven development (MDD) is to automate life cycle tasks using metadata collected by models. In MDD, models are not used primarily as documentation, but as source artifacts that can be compiled to produce implementation components, or used by other tools to support trace, navigation, validation, analysis, refactoring, optimization and other operations. While MDD can be practiced by extending general purpose modeling languages like UML, it is most effective when practiced with highly focused Domain Specific Languages (DSLs) designed to solve specific problems, related to specific tasks, on specific platforms, and with custom tools developed to support them [3]. There are many familiar examples of DSLs, such as SQL, BEPL, HTML and WSDL. DSLs make solutions easier to understand and maintain, and generally improve agility by facilitating rapid iteration.

Architecture Frameworks

As noted by Peter Deutsch, interface design and functional factoring constitute the key intellectual content of software, and are far more difficult to create or recreate than code. Architecture frameworks capture that intellectual content using viewpoints that identify, separate and interrelate well-defined sets of stakeholder concerns. Well known examples of architecture frameworks include the Zachman grid [4] and the TOGAF grid [5]. Examples of

stakeholders include business analysts, project managers, developers, and database administrators. Examples of viewpoints include business rules, business events, business information, business processes, user interface workflow, and database design. Examples of stakeholder concerns include how business rules are enforced by a given business process, or how a given physical database design supports a logical database design.

The software factory schema is an architecture framework. Unlike a grid, however, it is non-rectangular, allowing it to support nesting viewpoints, and to describe relationships among non-adjacent viewpoints. Also unlike a grid, it is specific to the solution family targeted by the factory, not generic. Finally, unlike a grid, the schema is dynamic, so that it can change to accommodate variations among the members of the solution family. For example, new viewpoints are added to the schema, and existing viewpoints are modified, when the Sales and Campaigns feature is added to an ecommerce application.

In addition to a set of stakeholder concerns, a viewpoint in a software factory schema defines a set of related artifacts relevant to those concerns, the activities that act upon the artifacts, and the assets used to perform the activities. The schema organizes the factory, and uses the relationships among viewpoints to integrate the activities, artifacts and assets across the software architecture and life cycle. For example, the relationships defined by the schema support operations like tracing features from requirements to implementation, navigating among solution artifacts, validating a deployment configuration across a set of connected systems, providing intellisense in user interface development from information captured during business information modeling, or generating a set of data access classes from a logical database design.

Using Software Factories

With a factory, projects start from well-defined baselines, instead of from scratch. Practitioners know what tasks need to be performed, what assets to use to perform each task, and how and why to use each asset. Guidance is available at their fingertips within the development environment. Junior developers follow guidance with minimal hand holding. Senior developers refine the guidance based on experience. The guidance is packaged and versioned, making it easier to gather feedback from users to improve its quality.

Development proceeds both top down, from viewpoints focusing on business goals and requirements and bottom up, from viewpoints focusing on implementation, testing, and deployment, toward viewpoints focusing on analysis and design. Tasks come into focus as the solution evolves, allowing the workflow to be driven by changes in circumstances, requirements and technologies, subject to correctness constraints, instead of following a prescriptive plan that may not be able to accommodate the realities of real world projects. Relationships between viewpoints are used to support the flow of development, through operations like trace, navigation, validation, analysis, refactoring, optimization and synchronization. In many cases, synchronization will involve generating some or all of the solution structure and contents targeted by viewpoints being synchronized. In other cases, it will involve editing the artifacts by hand to bring them fully into alignment.

A key feature of the methodology is that the products of two or more factories can be composed. Instead of a single factory that helps them build the ultimate deliverable in its entirety, users can work with multiple factories, each helping them build a portion of the ultimate deliverable.

The results are generally significant reduction in cost and time to market, significant improvements in quality attributes, and greater consistency from solution to solution. Risk is also reduced, as metric based estimation models can be constructed for the factory schema using

data collected during solution development, making it easier accurately forecast budget, schedule, and resource requirements, and to analyze the impact of new requirements. The time and cost of training new developers is also reduced.

When it is time to maintain or evolve a solution developed using a factory, the metadata captured during its development is used to analyze the impact of changes in requirements and technologies, and orchestrate the necessary changes. It is also used when underlying technology evolves to identify customizations made during solution delivery that must be migrated to the new technology, and often to fully or partially automate the migration of affected artifacts.

Mass Customization

Like other methodologies, Software Factories seek to improve productivity and predictability within a single organization. Unlike other methodologies, however, they also seek to support mass customization by promoting the formation of supply chains. Mass customization is the production of custom solutions on a large scale [6]. It is the result of bridging the gap between the diversity of customer requirements and economies of scale, and has been achieved in other industries, such as computers and automobiles. For example, a well known automobile manufacturer builds about 800 cars and 1200 engines per day at one of their factories. Each unit is built entirely to order. Due to the large number of options and the large number of valid ways in which those options can be combined, there are over 1 trillion possible variants of every product. It takes about 4 years, on average, to produce 2 units with exactly the same features.

Using Supply Chains

Packaged applications and custom development are the prevailing strategies for mass customization in the software industry. As we have seen, both have succeeded to some extent, but both have significant shortcomings.

Supply chains provide a more efficient and more economically viable approach to mass customization. Instead of relying on a single product based solution from a single vendor, suppliers at every level of the supply chain can select the best offerings from other suppliers to use in their product offerings. Customers can select from among the offerings at the surface of the supply chain those that best satisfy their unique and proprietary requirements. Of course, these products can then be tailored further by the suppliers or by service providers to create solutions customized to the needs of the individual customer.

As the products, the platform technologies, and the customer’s business needs change over time, the supply chain creates competitive pressures among its members to either adapt their offerings or be displaced by other suppliers who will offer what the supply chain demands. With solution development distributed over a large number of potential suppliers, the supply chain is much more responsive than even the best single vendor could ever be to changing technologies and market conditions. Cost, risk and time to market are also reduced at every level of the supply chain by the distribution of solution development and the competitive pressures it creates, resulting in significant savings to the customer.

The Alignment Problem

Currently, supply chains do not form rapidly in the software industry, and those that do form are generally much shallower than supply chains seen in other industries. Analysis suggests that the root cause is misalignment among suppliers. Different suppliers generally take different approaches to development. They have different ways of defining requirements, different ways

of organizing the software, different ways of using platform technologies, different ways of deploying system components, different ways of testing those components, and different ways of customizing system features. These differences, in turn, make it hard for others to integrate products from multiple suppliers to create solutions. As system integrators know, it is hard to design well formed solutions using product from multiple suppliers, hard to verify those solutions, hard to determine impact of changes in requirements or technologies, hard to determine what customizations were made for a given customer, hard to migrate customizations when the underlying products change, and hard to coordinate among suppliers to deliver custom features.

Supply Chain Formation

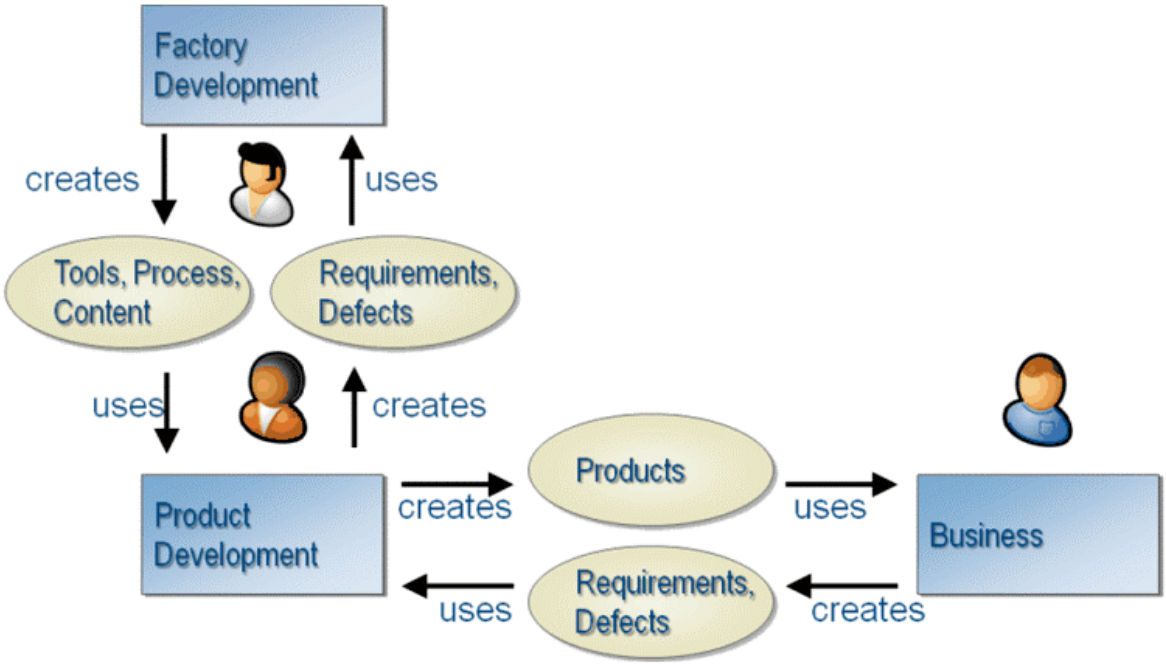
Software factories help to solve the alignment problem by making it easy to expose critical information about products in computationally convenient ways. This information helps suppliers to detect and address misalignment, and to globally optimize their supply chains.

Applying Factories

Replacing ad-hoc development with factory based development is the first step on the road to supply chains. Two important characteristics of factories, partitioning and assembly, make it technically feasible to distribute them across networks of cooperating organizations to facilitate the formation and operation of supply chains.

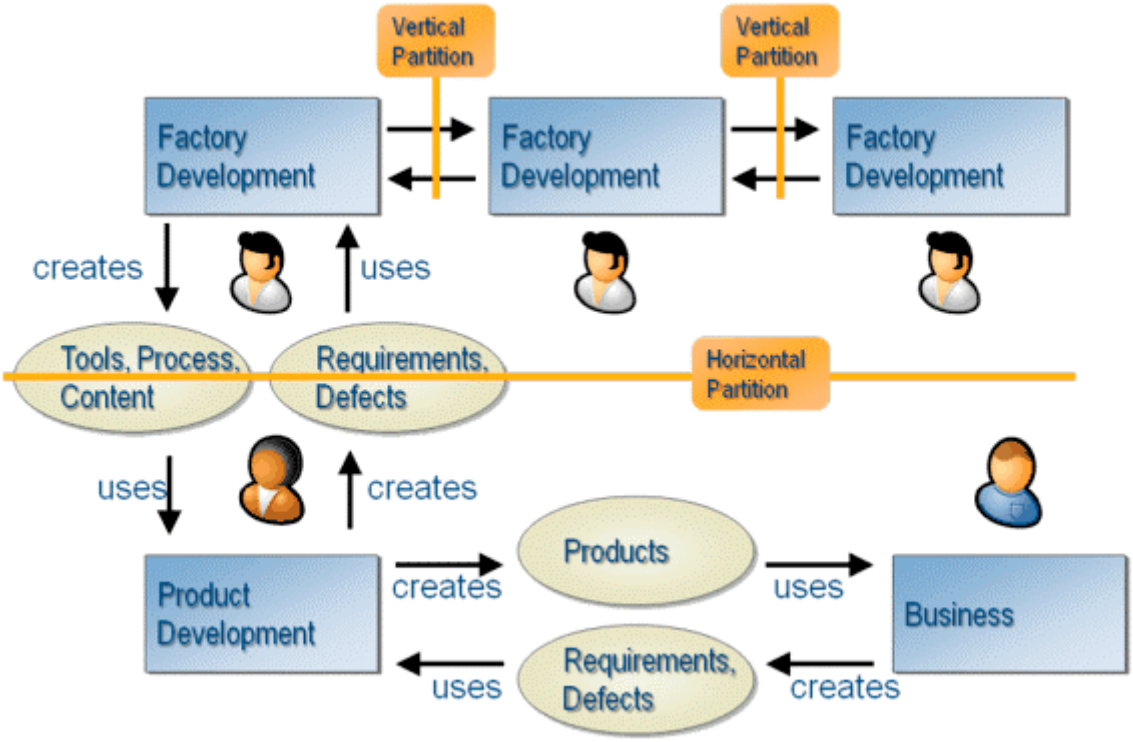
Factory Partitioning

A factory uses two interacting development processes. The first is the traditional development process by which solution developers build solutions for customers who use them to automate business processes, and who provide feedback to the solution developers, such as defect reports and feature requests.



The second is a separate and more specialized development process by which factory developers build assets for solution developers who use them to build solutions, and who provide feedback to the factory developers, such as defect reports and feature requests. These processes are illustrated in the diagram above.

Factories can be partitioned to form supply chains along these processes. They can be partitioned horizontally by placing the factory development process and the solution development process in separate organizations. For example, an Independent Software Vendor (ISV) might provide a factory used by Systems Integrators (SIs) to build solutions for customers. Alternatively, an enterprise might provide a factory used by an offshore service provider to build solutions for the enterprise. Factories can also be partitioned vertically by composition or specialization across the boundaries between organizations. For example, a banking self service portal factory might be developed by an SI by specializing a self service portal factory offered by an ISV, which might in turn be developed by specializing a portal factory offered by a platform vendor. The two types of partitions are illustrated in the following diagram.



As mentioned earlier, the factory schema provides metadata used to organize tools, processes and content by viewpoint, and to integrate them across relationships among viewpoints using operations like trace, navigation, validation, analysis, refactoring, optimization and synchronization. When a solution factory is partitioned across multiple suppliers, its schema is distributed across the supply chain, and the operations over relationships among the viewpoints support interactions among the suppliers, such as requirements communication, solution configuration, logical and technical architecture alignment, component assembly, adaptation and orchestration, workflow integration, test suite integration, and distributed deployment, management, maintenance and migration.

Factory Assembly

Factories can be composed and specialized. Smaller factories can be composed to form larger factories, and more general factories can be specialized to form more specialized factories. For

example, a factory for layered applications might be formed by composing factories for user interface, business logic and data access layers. A factory for retail banking self-service portals might be formed by specializing a factory for banking self-service portals, which might be formed by specializing a factory for self-service portals, which might be formed by specializing a factory for web portals.

Factory composition and specialization rely on the factory schema. Factory composition involves combining the viewpoints of the constituent factories, and factory specialization involves adding or removing viewpoints, and modifying viewpoints of the base factory by changing the artifacts produced, the activities that produce them, and the assets used to support and automate the activities.

Publishing Metadata

The second step on the road to supply chains is publishing information about key processes and artifacts, so that it can be consumed by other suppliers. This metadata is often readily available in the models used by factories. Two kinds of information are particularly important in supply chain formation:

- Information about requirements, such as the information captured by feature models [8] can be used to reason about compatibility, variability, and dependencies among requirements.
- Information about architecture, such as the information captured by service contracts, service level agreements and deployment manifests, can be used to reason about how products from other suppliers should be deployed, what platform services they require, and how they can be adapted, assembled and configured.

Both types of information help suppliers identify products that they might incorporate from other suppliers, and what changes they might request from them.

With any type of metadata, the key is making it readily available to other suppliers in computationally convenient ways. A relatively recent development that facilitates sharing information across organizational boundaries is web service technology. Using web services, suppliers can expose descriptions of externally facing artifacts and processes in a secure, reliable and platform independent manner. A promising application of web service technology from a supply chain perspective is composing services hosted by other suppliers to form new services, instead of composing components they supply to form new components. This kind of composition is commonly called a mash up, and enables some compelling scenarios, such as assembly by orchestration and automatic adaptation to address architectural mismatches.

Alignment and Optimization

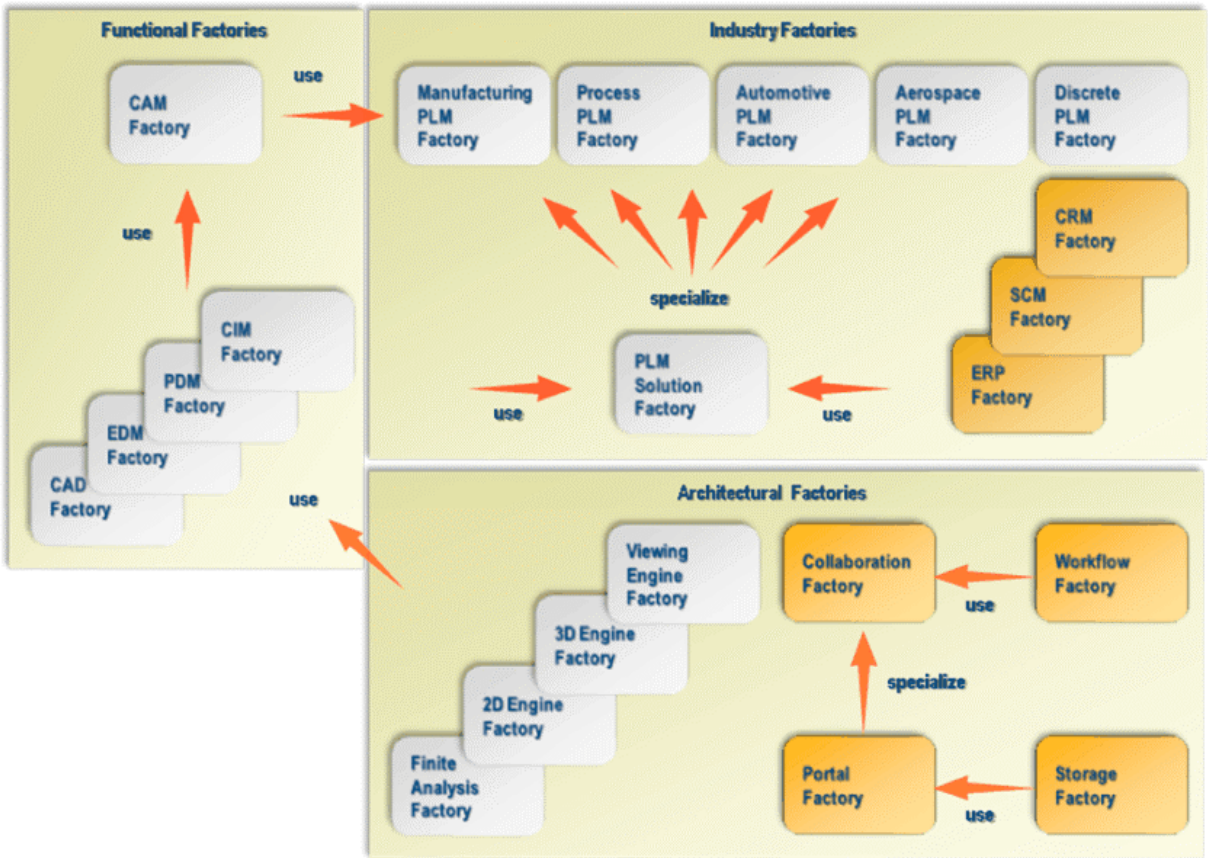
Publishing metadata is a necessary condition for supply chain formation, but not a sufficient condition. The final step on the road to supply chains involves discovering and correcting misalignments among suppliers using the published metadata, and optimizing the resulting network. Aligned suppliers have agreed to use compatible technologies, and have identified effective ways to integrate their products. In the real world, these alliances are constantly renegotiated as technologies and business conditions change, and suppliers enter and leave the marketplace.

When the suppliers are aligned, we can turn our attention to optimizing the resulting supply chain. The goal of optimization is to eliminate bottlenecks that reduce the speed at which changes can propagate through the network. Optimization generally involves replacing local

responses to changes by individual suppliers with collective responses by reasoning over published dependency information. Without optimization, suppliers individually recognize and respond to changes in products from other suppliers as they encounter them. Each change therefore propagates incrementally from its source to its ultimate consumers, one supplier at a time. With optimization, information about changes that might affect large numbers of downstream suppliers can be published as the changes arise, allowing downstream suppliers to anticipate and respond to the changes before they arrive. This kind of optimization is common in supply chains in other industries, such as computers and automobiles.

Conclusion

The following diagram illustrates a supply chain for Product Lifecycle Management (PLM) applications assembled from factories. This kind of industry architecture and the efficient mass customization it enables is the ultimate goal of the Software Factories methodology.



While Software Factories provide the technical and architectural foundation required to enable the formation of supply chains in the software industry, they are not enough to change the market dynamics by themselves. Business models that promote and facilitate supply chain formation and that enable efficient supply chain operation are also required. A business model discussion is beyond the scope of this paper.

References

1. Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley. ISBN: 0-471-20284-3
2. Paul Clements and Linda Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley Professional. ISBN-10: 0201703327
3. Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional. ISBN-10: 0321398203
4. John Zachman . The Framework for Enterprise Architecture: Background, Description and Utility. Zachman Institute for Framework Advancement (ZIFA). Document ID: 810-231-0531
5. www.opengroup.org/togaf
6. Joseph Pine. Mass Customization: The New Frontier in Business Competition. Harvard Business School Press. ISBN: 0-87584-946-6
7. David Anderson. Build-to-Order & Mass Customization. CIM Press. ISBN: 1-878072-30-7
8. Krzysztof Czarnecki and Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional. ISBN-10: 0201309777

Free White Paper from MKS - From Ad Hoc to Optimized: Automating IT and Development Processes for Agility and Efficiency. Do you find that you are making changes to applications on the fly? Are you using paper-based or time-consuming manual IT processes? Is it difficult to show the history of software change to an auditor? Make the move from ad hoc to optimized process with MKS Integrity for application lifecycle management (ALM). Download a free white paper and learn how:

http://www.mks.com/mtoptimizeprocesswp

While the principles of the Manifesto for Agile Software Development may look appealing for inexperienced developers, serious professionals know that the real world is not similar to the "Little House on the Prairie". Look at this humourous mirror site to the Manifesto for Agile Software Development

http://www.waterfallmanifesto.org

The Software Quality Assurance Zone is a repository for resources concerning software testing (unit testing, functional testing, regression testing, load testing), code review and inspection, bug and defect tracking, continuous integration. The content consists of articles, news, press releases, quotations, books reviews and links to articles, Web sites, tools, blogs, conferences and other elements concerning software quality assurance. Feel free to contribute with your own articles, links or press releases.

http://www.sqazone.net/

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This clasified section is waiting for you at the price of US $ 30 each line. Reach more than 47'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 30'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.html

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php
The content of this publication cannot be reproduced without prior written consent of the publisher
Copyright © 2007, Martinig & Associates