METHODS & TOOLS

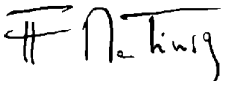
Global knowledge source for software development professionals **ISSN 1661-402X**

Summer 2006 (Volume 14 - number 2) www.methodsandtools.com

Anyone Heard About 5th Generation Languages?

Borland has decided to sell its IDE business that has produced some good material for programmers like Turbo-Pascal or Delphi. Some people were asking what the future of Delphi would be. This point nourished my recent thoughts on the evolution of programming languages. In the 80's, when languages were the main source for marketing material, the term "4th generation language" (4GL) was used to promote development environments where you could program applications quicker than with "traditional" 3rd generation languages like C or Cobol. They were based on "high-level" products like Natural, Focus, Nomad or Progress. Many of these products have now a limited usage, were abandoned or were bought by Computer Associates. I let you choose if there is a difference between options 2 and 3. Some named artificial intelligence (AI) languages, like Lisp or Prolog, as 5th generation languages, but I disagree with this. So 4GL could be considered as an "ultimate" evolution of programming languages. Why?

A new orientation in programming occurred already when the mainstream focus changed to object orientation and API. Now in the era of Web 2.0 (you could compete for the "Dinosaur of the month" title by saying that you will wait for Web 2.1, because it will be less buggy), most web pages are build mixing several interpreted languages (Java, PHP, SQL, JavaScript, HTML, Python, etc.), even with some contribution from C or Cobol modules. Many trends are behind this. First the "write once, run everywhere" paradigm forced the switch from a compiled world to interpreters to gain independence from hardware. The management of languages is also different. The new languages are now developed with a more or less important participation of the programming community. Flash could be considered the sole exception of a new widely diffused proprietary language, outside the Microsoft world. Finally, the new languages are mainly specialised. They don't try to manage everything from the user interface to the database, but instead they focus on server or client-side operations. Fourth generations languages could have been the last attempt to lock programming in proprietary full service development environment. You can now say "Hello World" without thinking about generations.



Inside

Project Failure Prevention: 10 Principles for Project Control

Tom Gilb, Tom@Gilb.com
www.gilb.com

Abstract: It is now well-known and well-documented that far too many projects fail totally or partially, both in engineering generally (Morris 1998) and software engineering (Neill and Laplante 2003). I think everybody has some opinions about this. I do too, and in this paper I offer some of my opinions, and I hope to lend some originality to the discussion. As an international consultant for decades, involved in a wide range of projects, and involved in saving many 'almost failed' projects, my basic premises in this paper are as follows:

- We specify our requirements unclearly;
- We do not focus enough on ensuring that the system design meets the requirements.

INTRODUCTION

The principles for project control can be summarized by a set of ten principles, as follows:

P1: CRITICAL MEASURES: The critical few product objectives (performance requirements) of the project need to be stated measurably.

P2: PAY FOR RESULTS: The project team must be rewarded to the degree they achieve the critical product objectives.

P3: ARCHITECTURE FOR QUALITY: There must be a top-level *architecture* process that focuses on finding and specifying appropriate design strategies for enabling the critical product objectives (that is, the performance requirements' levels) to be met on time.

P4: CLEAR SPECIFICATIONS: Project specifications should *not* be polluted with dozens of defects per page; there needs to be specification quality control (SQC) with an exit condition set that there should be less than one remaining major defect per page.

P5: DESIGN MUST MEET THE BUSINESS NEEDS: Design review must be based on a 'clean' specification, and should be focused on whether the designs meet the business needs.

P6: VALIDATE STRATEGIES EARLY: The high-risk strategies need to be validated early, or swapped with better ones.

P7: RESOURCES FOR DESIGNS: Adequate resources need to be allocated to deliver the design strategies.

P8: EARLY VALUE DELIVERY: The stakeholder value should be delivered early and continuously. Then, if you run out of resource unexpectedly, proven value should already have been delivered.

P9: AVOID UNNECESSARY DESIGN CONSTRAINTS: The requirements should not include unnecessary constraints that might impact on the delivery of performance and consequent value.

P10: VALUE BEFORE BUREAUCRACY: The project should be free to give priority to value delivery, and not be constrained by well-intended processes and standards.

PRINCIPLES

P1: CRITICAL MEASURES: The critical few product objectives (performance requirements) of the project need to be stated measurably.

The major reason for project investment is always to reach certain levels of product performance. 'Performance' as used here, defines how good the system function is. It includes:

- *Qualities* - how well the system performs;
- *Resource Savings* - how cost-effective the system is compared to alternatives such as competitors or older systems;
- *Workload Capacity* - how much work the system can do.

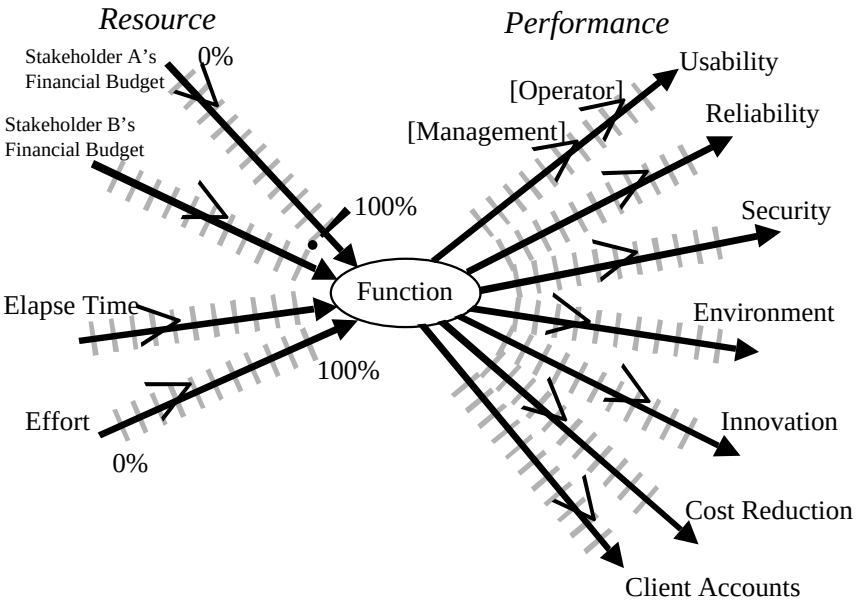


Figure 1. The 'product' of a project will want to attain a number of critical performance requirements. Serious project control necessitates clear agreement about the set of performance levels. The project can then focus on delivering these levels, within the available resources

In practice, you need to be concerned with the 5 to 20 'most-critical' product performance requirements (For example, see Figure 1). These are the critical dimensions that determine if a project has been a success or failure, in terms of the *product* produced by that project. I choose to make a clear distinction between the *project* characteristics (like team spirit and budget overrun) and the project *product* characteristics, and to focus here on the *product* characteristics as the decisive success or failure concepts. I am not concerned with 'the operation was a success, but the patient died' view of systems engineering.

I observe, in project after project, that I almost never see what I would call a well-written set of top-level requirements. The problems, which I perceive, include:

- The critical product characteristics are often not clearly identified *at all;*
- They are often identified only in terms of some *proposed design* (like 'graceful file degradation' to quote a recent one) to achieve requirements (rather than 'file availability', a requirement area);

- They are often pitched at an inappropriately *technical* level ('modularity' rather than 'flexibility');
- When they *are* identified they are often specified in terms of *'nice words'* (for example, 'state-of-the-art security') rather than a *quantified* engineering specification (such as '99.98% reliability');
- Even when some quantification is given - it often lacks *sufficient detail and variety* to give engineering control. For instance including the *short*-term goals – not just the final goals, and including the different goals for the variety stakeholders – not just the implied system user. I usually see no explicit statement of the rationale for the performance levels specified.

If the critical success factors for the projects output are not well specified, then it does not matter how good any consequent process of design, quality control, or project management is. They cannot succeed in helping us meet our primary product requirements. See Figure 2 for an example of a quantitative specification of a performance requirement. This is the level of detail that I consider appropriate.

Requirement Tag: Interoperability:

Interoperability: defined as: The ability of two or more IS, or the subcomponents of such systems, to exchange information and services, and to make intelligent use of the information that has been exchanged < JSP.

Vision: The system shall make business application data visible across the boundaries of component sub-systems <- SRS 2.2.7.

Source: SRS Product ID [S.01.18, 2.2.7].

Version: October 2, 2001 11:29.

Owner: Mo Cooper.

Ambition: Radically much better Interoperability than previous systems.

Scale: Seconds from initiation of a defined [Communication] until fully successful intended intelligent [Use] is made of it, under defined field [Conditions] using defined [Mode].

Meter [Acceptance] <A realistic range of at least 100 different types of Communication and 100 Use and 10 Conditions> <- TG.

=== Benchmarks ========== *Past Levels* ==========

Past [UNNICOM, 2001, Communication = Email From Formation to Unit, Use = Exercise Instructions, Conditions = Communication Links at Survival]: <infinite> seconds <- M Cxx.

Conditions: defined as: Field conditions, which might threaten successful use.

Record [DoD, 1980?, Communication = Email From Formation to Unit, Use = Exercise Instructions, Conditions = Communication Links at Survival]: 5 seconds <- ??

Trend [MoD UK, IT systems in General, 2005, Mode = {Man transporting paper copy on motorbike, or any other non-electronic mode}]: 1 hour?? <- TG.

=== Targets ========== *Required Future Levels* ==========

Goal [DoD, 2002, Communication = Email From Formation to Unit, Use = Exercise Instructions, Conditions = Communication Links at Survival]: 10 seconds?? <- ??

Figure 2. Specifying a performance requirement using Planguage. This example is a first draft from a real project

P2: PAY FOR RESULTS: The project team must be rewarded to the degree they achieve the critical product objectives.

What do we do for the project team when they *fail* to deliver the specified requirements, or when they use more time and money than budgeted? We *reward* them by continuing to pay them.

Now, if being out of control on performance and costs was entirely beyond their powers, then ‘payment for effort’ might be a reasonable way to do things. But I believe that we would be much better off if there were clear rewards for reaching targeted performance levels within budgeted resources. And lack of reward, for failing.

Advertisement – Code Collaborator - Click on ad to reach advertiser web site

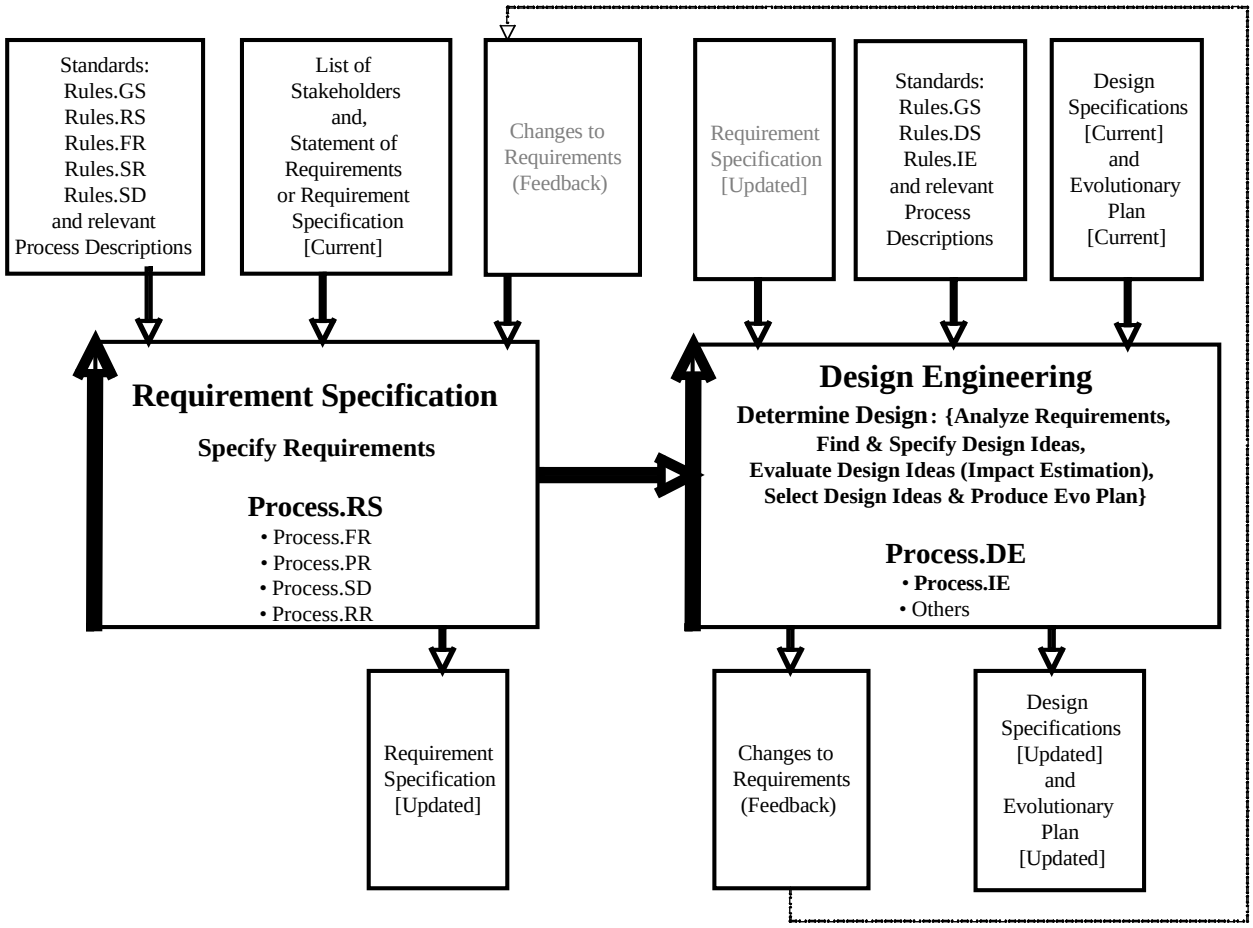


Figure 3. The design engineering process depends on the quality of the requirement specification. The standard for the requirement specification can be set partly by the specification rules, and partly by the defect density tolerance (that is, how seriously the rules are taken as measured by a quality control process). The standard processes and tags (for example, 'Rules.GS') refer to the standards published in the CE manuscript (Gilb 2005)

We would, of course, have to make it a 'fair' game. The project team would have to voluntarily agree that the performance goals were in fact realistic, in relation to the resources they have, or can get, to deliver them. We would also need to resist the temptation to dictate work processes or to specify designs, which in the view of the project team, would stop them from achieving the project goals. Of course if we still cannot specify the performance goals quantitatively, then no amount of motivation and freedom will get a project team to move in the right direction. They don't even know what that direction is.

P3: ARCHITECTURE FOR QUALITY: There must be a top-level *architecture* process that focuses on finding and specifying appropriate design strategies for enabling the critical product objectives (that is, the performance requirements' levels) to be met on time.

If you do not have a clear quantified set of top-level critical requirements, then an architecture process is bound to fail. The architect cannot compare *their* design idea's expected performance and cost attributes with the clear performance and cost *requirements*, they need to have for clear judgements about design ideas. Even if the requirements are perfectly quantified and clear, that is not sufficient. The architecture designs themselves must be specified in sufficient detail to enable an judgement that they will probably provide the necessary levels of performance and cost impacts (See Figure 3).

Tag: OPP Integration.

Type: Design Idea [Architectural].

========== Basic Information ==========

Version:

Status:

Quality Level:

Owner:

Expert:

Authority:

Source: System Specification [Volume 1 Version 1.1, SIG, February 4. – Precise reference <to be supplied by Andy>].

Gist: The X-999 would integrate both 'Push Server' and 'Push Client' roles of the Object

Push Profile (OPP).

Description: Defined X-999 software acts in accordance with the <specification> defined for both the Push Server and Push Client roles of the Object Push Profile (OPP). Only when official certification is actually and correctly granted; has the {developer or supplier or any real integrator, whoever it really is doing the integration} completed their task correctly. This includes correct proven interface to any other related modules specified in the specification.

Stakeholders: Phonebook, Scheduler, Testers, <Product Architect>, Product Planner, Software Engineers, User Interface Designer, Project Team Leader, Company engineers, Developers from other Company product departments, which we interface with, the supplier of the TTT, CC. "Other than Owner and Expert. The people we are writing this particular requirement for"

========== Design Relationships ==========

Reuse of Other Design:

Reuse of this Design:

Design Constraints:

Sub-Designs:

========== Impacts Relationships ==========

Impacts [Intended]: Interoperability.

Impacts [Side Effects]:

Impacts [Costs]:

Impacts [Other Designs]:

Interoperability: Defined As: Certified that this device can exchange information with any other device produced by this project.

========== Impact Estimation/Feedback ==========

Impact Percentage [Interoperability, Estimate]: <100% of Interoperability objective with other devices that support OPP on time is estimated to be the result>.

========== Priority and Risk Management ==========

Value:

Figure 4. An example of a real draft design specification that attempts to both have necessary detail, and to make some assertions and estimates about the effects of the design on requirements. Much more could be specified later.

P4: CLEAR SPECIFICATIONS: Project specifications should not be polluted with the usual dozens of defects per page; there needs to be specification quality control (SQC) with an exit condition set that there should be less than one remaining major defect per page.

I regularly find that any requirement specification given to me by a new customer, even if it is approved and being used, has between 80 and 180 'major' defects. This is normally a 'shock' for the people involved. How can there be so many? We measure them by asking colleagues of the specification author, and often the author too, to check a sample 'logical' page (that is, 300 non-commentary words) of a requirements specification, and count any violations of these simple rules:

- Clear enough to test;
- Unambiguous to the intended readership;
- No unintentional design in the requirements.

I then ask participants to evaluate if each rule violation (defect) is serious enough to potentially cause delays, costs, and product defects. They are asked to classify any that are serious, as 'major' defects. The range of majors found per sample page, in about 15 minutes of checking, is usually from 3 to 23 per person. I find that small groups of 3 to 4 people typically find about double the most defects found by a single individual. For example, if the greatest number of defects found by one person is 15, then a small group would have about 30 unique majors to report. But these 30 are only one third of what is actually in the spec right now. The checking process is about 30% effective. Consequently there are something like 90 major defects present in the page, of which the small group can find only a third in about 15 minutes.

Most people immediately agree that this is far too many. It is. And it is unnecessary! How polluted a requirements specification would you think is acceptable? If you are professional you will finally set a limit of *no more than one remaining major defect per page* before you release the specification for use in design or test planning.

Total Defects M+m	**Majors M**	**Design (part of Total and M+m) Design**
41	24	D=1
33	15	D=5
44	30	D=10
24	3	D=5

Table 1: An example from a 30 minute checking of real project requirements at a Jet Engine Company by 4 managers. The sample page was called 'Non-Functional requirements. By extrapolation the team found about 60 of a total in the page of about 180 major defects. The 'M+m' is majors and minors. The 'D' numbers are the number of designs in the requirements

My experience (since 1988 working with aircraft drawings in California) is that if you set such exit conditions (max. 1 major/page) and take them seriously, then individuals will learn to reduce their personal injection of major defects by about 50% for each SQC cycle they go through. Within about 7 cycles, the individual engineer will be able to specify such a clean specification that it will exit first time. Some people can achieve this with fewer cycles.

The conclusion is that because we do not carry out even simple inexpensive sampling measures of specification pollution, and we do not set a limit to the pollution, we live in a world of engineering that is normally highly polluted. We pay for this by project delays, and poor product quality.

P5: DESIGN MUST MEET BUSINESS NEEDS: Design Review must be based on a 'clean' specification, and should be focused on whether the designs meet the business needs.

Before we can review a design specification regarding relevance, we must review it for 'compliance to rules of specification', that assures us about the *intelligibility* of the specification.

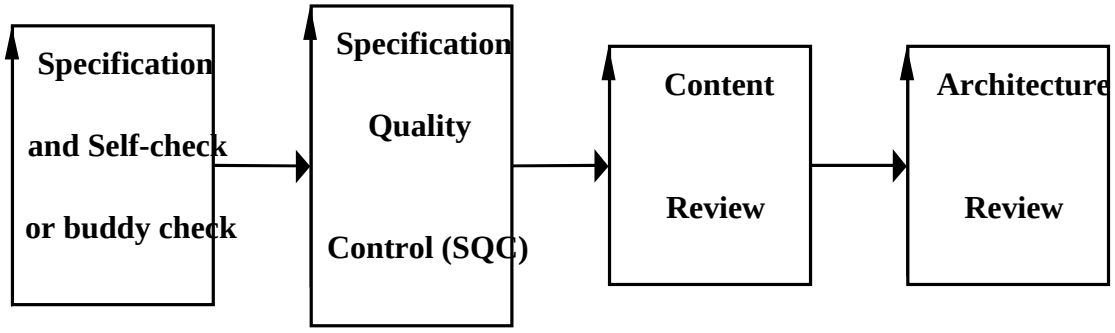


Figure 5. Four stages of checking a design specification

For example, there is no point in reviewing a design specification (such as 'Reusable Modules'), when

- The specification is unnecessarily ambiguous;
- There is no assertion of which requirements the design is supporting;
- There is no assertion or estimation as to how well the design is supporting those requirements;
- There are no resource estimates (costs) for the design.

To put it more directly: Check that a design specification is *well written* first, only then do you have a basis for checking to see if it is a *good* design. I believe it is unreasonable to judge a design idea itself on the basis of a poorly written specification. Only well-specified designs should be evaluated by senior responsible people. Badly specified designs – defined as those not following our own design specification rules – need to be returned to the designer for rework. See Figure 6 for an example of real design rules designed to force us to specify designs in sufficient detail. When these rules are followed, then, I believe we have the basis for deciding whether a design is appropriate in relation to its requirements.

Rules: Design Specification

Tag: Rules.DS. Version: October 7, 2004. Owner: TG. Status: Draft.

Note: Design specifications are either for optional design ideas (possible solutions) or required design constraints (that is, actual requirements AND consequently, pre-selected solutions).

Base: The rules for generic specification, Rules.GS apply. If the design idea is a design constraint (a requirement), the rules for requirement specification, Rules.RS also apply.

R1: **Design Separation**: Only design ideas that are intentionally 'constraints' (*Type: Design Constraint*) are specified in the *requirements*. Any other design ideas are specified separately (*Type: Design Idea*). Note all the design ideas specified as requirements should be explicitly identified as 'Design Constraints.'

R2: **Detail**: A design specification should be specified in *enough detail* so that we know precisely what is expected, and do not, and cannot, inadvertently assume or include design elements, which are not actually intended. It should be 'foolproof'. For complex designs, the detailed definition of its sub-designs can satisfy this need for clarity, the high level design description does not need to hold all the detail.

R3: **Explode**: Any design idea, whose impact on attributes can be better controlled by detailing it, should be broken down into a list of the tag names of its elementary and/or complex sub-design ideas. Use the parameter 'Definition' for Sub-Designs. If you know it can be decomposed; but don't want to decompose it just now, at least explicitly indicate the potential of such a breakdown. Use a Comment or Note parameter.

R4: **Dependencies**: Any known dependencies for successful implementation of a design idea need to be specified explicitly. Nothing should be assumed to be 'obvious'. Use the parameter, Dependency (or Depends On), or other suitable notation such as [qualifiers].

R5: **Impacts**: For each design idea, specify *at least one* main performance attribute impacted by it. Use an impact arrow '->' or the Impacts parameter. Comment: At early stages of design specification, you are just establishing that the design idea has some relevance to meeting your requirements. Later, an IE table can be used to establish the performance to cost ratio and/or the value to cost ratio of each design idea.

Example:

Design Idea 1 -> Availability.

Design Tag 2: Design Idea.

Impacts: Performance X.

R6: **Side Effects**: Document in the design specification any side effects of the design idea (on defined requirements or other specified potential design ideas) that you expect or fear. Do this using explicit parameters, such as Risks, Impacts [Side Effect] and Assumptions.

Examples:

Design Idea 5: Have a <circus> -> Cost A.

Risk [Design Idea 5]: This might cost us more than justified.

Design Idea 6: Hold the conference in Acapulco.

Risk: Students might not be able to afford attendance at such a place?

Design Idea 7: Use Widget Model 2.3.

Assumption: Cost of purchasing quantities of 100 or more is 40% less due to discount.

Impacts [Side Effects]: {Reliability, Usability}.

Do not assume others will know, suspect or bother to deal with risks, side effects and assumptions. Do it yourself. Understanding potential side effects is a sign of your system engineering competence and maturity. Don't be shy!

R7: **Background Information**: Capture the Background information for any estimated or actual *impact* of a design idea on a performance/cost attribute. The evidence supporting the impact, the level of uncertainty (the error margins), the level of credibility of any information and, the source(s) for all this information should be given as far as possible. For example, state a previous project's experience of using the design idea. Use Evidence, Uncertainty, Credibility, and Source parameters. Note the source icon (<-) usually represents the Source parameter. Comment: This helps 'ground' opinions on how the design ideas contribute to meeting the requirements. It is also preparation for filling out an IE table.

Example:

Design Tag 2 -> Performance X <- Source Y.

R8: **IE Table**: The set of design ideas specified to meet a set of requirements should be validated at an early stage by using an Impact Estimation (IE) table.

Does the selected set of design ideas produce a good enough set of expected attributes, with respect to all requirements and any other proposed design ideas? Use an IE table as a working tool when specifying design ideas and also, when performing quality control or design reviews on design idea specifications.

R9: **Constraints**: No single design specification, or set of design specifications cumulatively, can violate any specified constraint. If there is *any* risk that this might occur the system engineer will give a suitable warning signal. Use the Risk or Issues parameters, for example.

R10: **Rejected Designs**: A design idea may be declared 'rejected' for any number of reasons. It should be retained in the design documentation or database, with information showing that it was rejected, and also, why it was rejected and by whom.

Example:

Design Idea D: Design Idea.

Status: Rejected.

Rationale [Status]: Exceeds Operational Costs.

Authority: Mary Fine. Date [Status]: April 20, This Year.

Figure 6. Specification rules for design, which lay a proper basis for judging the power and cost of the design in relation to the requirements. See also Chapter 7 in (Gilb 2005)

P6: VALIDATE STRATEGIES EARLY: The high-risk strategies need to be validated early, or swapped with better ones.

It should be possible to identify your high-risk strategies (Note, 'strategies' are also known as designs, solutions and architectures). They are the ones that you are not sure of the impact levels on performance and cost. They are the ones that can potentially ruin your project, if they turn out to be worse than you are expecting. If you try to estimate all impacts of a given strategy on an Impact Estimation table, and get estimates such as 50%±40% (100% = Goal level attained), then you have exposed a high-risk design. If the credibility estimate (a defined Impact Estimation method) on a scale of 0.0 to 1.0 is low, like under 0.5, then you also have a high-risk strategy. If no one will guarantee the result in a binding contract, then you also have a high-risk strategy.

Engineers have always had a number of techniques for validating risky strategies early. Trials, pilots, experiments, and prototypes are common tactics. One approach I particularly favor is scheduling the delivery of the risky strategy in an early evolutionary delivery step, to measure what happens – and thus get rid of some of the risk. Early evolutionary delivery steps usually integrate a new strategy with a real system, and with real users, and are therefore more trustworthy, than, for example, an expert review panel, which is relatively theoretical.

Policies for Risk Management

Explicit Risk Specification: All managers/planners/engineers/testers/quality assurance people shall specify any uncertainty, and any special conditions, which can imaginably lead to a risk of deviation from defined target levels of system performance. This must be done at the earliest opportunity in writing, and integrated into the main plan.

Numeric Expectation Specification: The expected levels of all quality and cost attributes of the system shall be specified in a numeric way, using defined scales of measure, and at least an outline of one or more appropriate 'meters' (that is, a proposed test or measuring instrument for determining where we are on a scale).

Conditions Specified: The requirements levels shall be qualified with regard to when, where and under which conditions the targets apply, so there is no risk of us inadvertently applying them inappropriately.

Complete Requirement Specification: A *complete* set of all critical performance and cost aspects shall be specified, avoiding the risk of failing to consider a single critical attribute.

Complete Design Specification and Impact Estimation: A complete set of designs or strategies for meeting the complete set of performance and cost targets will be specified. They will be validated against all specified performance and cost targets (using Impact Estimation Tables). They will meet a reasonable level of safety margin. They will then be evolutionarily

validated in practice before major investment is made. The Evo steps will be made at a rate of maximum 2% of total project budget, and 2% of total project timescales, per increment (Evo step) of designs or strategies.

Specification Quality Control, Numerically Exited: All requirements, design, impact estimation and evolutionary project plans, as well as all other related critical documents, such as contracts, management plans, contract modifications, marketing plans, shall be 'quality controlled' using the Inspection method (Gilb 1993). A normal process exit level shall be *that 'no more than 0.2 maximum probable major defects per page can be calculated to remain, as a function of those found and fixed before release, when checking is done properly'* (that is, at optimum checking rates of 1 logical page or less per hour).

Evolutionary Proof of Concept Priorities: The Evolutionary Project Management method will be used to sense and control risk in mid-project. Dominant features will include:

• 2% (of budget and time-to-deadline) steps;

• High value-to-cost steps, with regard to risk, prioritized asap;

• High risk strategies tested 'offline to customer delivery', in the 'backroom' of

the development process, or at cost-to-vendor, or with 'research funds' as opposed to using the official project budget.

Figure 7. Policy Ideas for Risk Management

P7: RESOURCES FOR DESIGNS: Adequate resources need to be allocated to deliver the design strategies.

It is not sufficient that your design strategies will meet your *performance* targets. We have to have the *resources* needed to implement them. And the resources used must not result in an unprofitable situation, even if we can get them. The resources we must consider are both for development and operation, even decommissioning. The resources are of many types, and include money, human effort and calendar time.

I observe that it is unusual for me to see design specifications with detailed cost calculations at the level of *individual designs*. We do not even seriously try to consider individual design idea costs (at best we estimate a total cost); so it is not surprising that we constantly exceed the resource constraints that apply to our projects.

P8: EARLY VALUE DELIVERY: The stakeholder value should be delivered early and continuously. Then, if you run out of resource unexpectedly, proven value should already have been delivered.

Projects may fail because they run out of time or money and have delivered nothing. They can then only offer hope of something, at the cost of additional money and time – and note that this addition is typically estimated by the people who have already demonstrated they do not know what they are promising.

Designs->	**Contract**	**Supplier**	**Motive**	**Architect**	**Parts Used**	*Sum* ***%Impacts***
Requirements						
Performance						
Quality 1	0%	100%	50%	30%	-20%	160%
Quality 2	100%	50%	0%	20%	50%	220%
Costs						
Investment Cost	5%	10%	1%	10%	110%	136%
Operational Cost	5%	50%	20%	1%	10%	86%
Staff Resource	10%	20%	10%	5%	0%	45%
Performance to Cost Ratio	100/20	150/80	50/31	50/16	30/120	

Table 2: An Impact Estimation table structure simplified example that helps us consider cost elements, while looking at the performance impacts. Consequently we can see the performance to cost ratio. The % estimates refer to % of meeting a performance level target, or to % of a budgeted cost

The smart management solution to this common problem is to demand that projects are done evolutionarily. That means there will be consciously-planned early (first month) and frequent (perhaps weekly, or 2% of budget) attempts to deliver measurable value to real stakeholders.

Most people have never been subject to this discipline, and they have not learned the theory of how to do it in practice. But there are decades of practical proof in the software and systems engineering world that this works. (Larman and Basili 2003, Larman 2003).

P9: AVOID UNNECESSARY DESIGN CONSTRAINTS: The requirements should not include unnecessary constraints that might impact the delivery of performance and consequent value.

It is all too common for projects to focus on a particular technical solution or architecture, and not to focus on the actual end results they expect to get from the technical 'design'. They end up locking themselves into the technical solution – and rarely get the results they expected. Remember the high failure rate of projects?

The primary notion in planning any project, and in contracting suppliers to deliver all or part of it, is to focus entirely on the top few critical results. The critical results have to be quantified within the requirements and made the subject of contract conditions (such as, 'No cure, No pay').

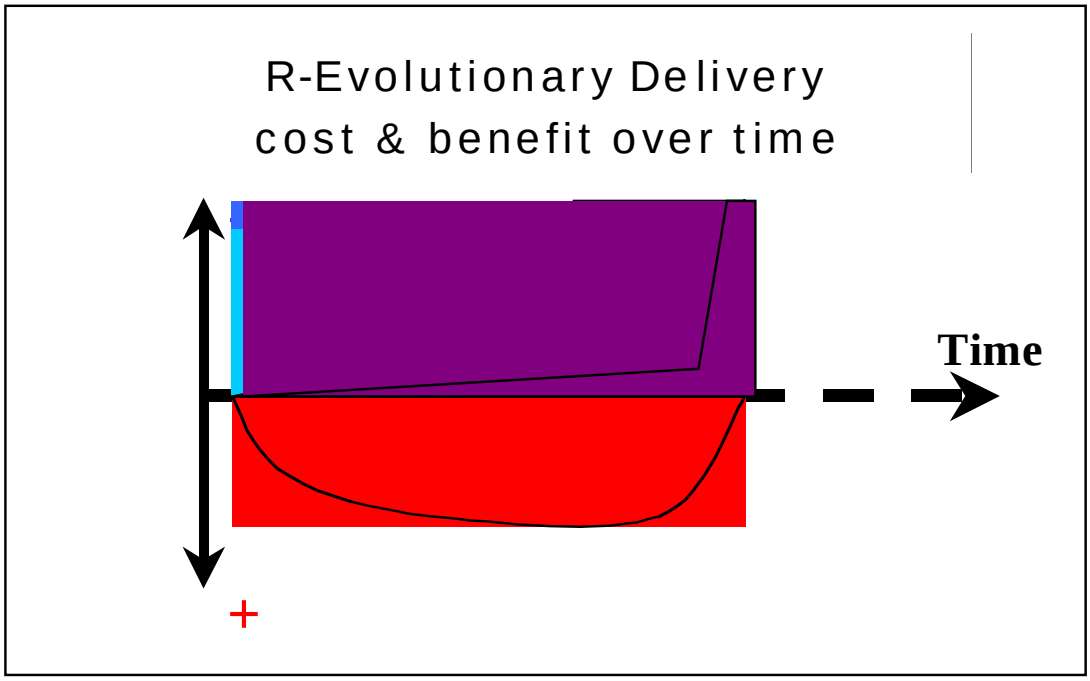


Figure 8. Conceptual view of delivery of stakeholder benefits early and cumulatively. From Kai Gilb's book Manuscript (Gilb, K. 2005)

Technology Policy

- Use the technology which best satisfies your requirements.
- If you are responsible for results, you have total discretion about the means.
- Those who dictate constraints are responsible for the constraint effects.
- Make sure the risk of technology you choose is understood, controlled and profitable.
- Satisfy stakeholder priorities in profitable sequence, within available resources.

Figure 9. A technology policy the author suggested to one of his clients, a major electronics telecom organization

P10: VALUE BEFORE BUREAUCRACY: The project should be free to give priority to value delivery, and not be constrained by well-intended processes and standards.

There was a time when software and IT were 'Wild West'. Anybody who could program, did things as they knew best. Sometimes, we are not far in many places from that model today. However in other places, the need to get higher consistent standards of professionalism, has swung the pendulum too far the other way. Processes and standards like the Software Engineering Institute Capability Maturity Model Integration (CMMI 2002) are thorough and well intended. But almost no such recommended frameworks and processes encourage or permit focus on the *main results* of a project. Consequently there is a great, inevitable, danger that this *results focus* will be lost in practice. Everywhere I look, I see that result – no results focus – worldwide – with or without the distraction of CMMI and the like. This includes the Agile Methods (Larman 2003). My recommendation to attempt to refocus is outlined in Figure 10.

A Simple Evolutionary Project Management Method

Project Process Description

1. Gather from all the key stakeholders the top few (5 to 20) most critical performance (including qualities and savings) goals that the project needs to deliver. Give each goal a reference name (a tag).

2. For each goal, define a scale of measure and a 'final' goal level. For example: *Reliability: Scale: Mean Time Between Failure, Goal: >1 month.*

3. Define approximately 4 budgets for your most limited resources (for example, time, people, money, and equipment).

4. Write up these plans for the goals and budgets (*Try to ensure this is kept to only one page*).

5. Negotiate with the key stakeholders to formally agree the goals and budgets.

6. Plan to deliver some benefit (that is, progress towards the goals) in *weekly* (or shorter) increments (Evo steps).

7. Implement the project in Evo steps. Report to project sponsors after each Evo step (weekly, or shorter) with your best available estimates or measures, for each performance goal and each resource budget.
- O*n a single page,* summarize the *progress to date* towards achieving the goals and the costs incurred.
- Based on numeric feedback, and stakeholder feedback; *change whatever needs to be changed to reach goals.*

8 When all goals are reached: "Claim success and move on" (Gerstner 2002). Free the remaining resources for more profitable ventures

Project Policy for Simple/Agile Evo Projects

1. The project manager, and the project, will be judged exclusively on the relationship of progress towards achieving the goals versus the amounts of the budgets used. The project team will do anything legal and ethical to deliver the goal levels within the budgets.

2. The team will be paid and rewarded for 'benefits delivered' in relation to cost.

3. The team will find their own work process, and their own design.

4. As experience dictates, the team will be free to suggest to the project sponsors (stakeholders) adjustments to 'more realistic levels' of the goals and budgets. For more detail, see (Gilb 2003, Gilb 2004).

Figure 10.

Acknowledgements

Article edited by Lindsey Brodie, lindseybrodie@btopenworld.com, Middlesex University. London

References

Bahill, A. Terry and Henderson, Steven J., "Requirements Development, Verification and Validation Exhibited in Famous Failures", *Systems Engineering*, Vol. 8, No. 1, 2005 pp. 1-14

CMMI, Capability Maturity Model Integration, Carnegie Mellon Software Engineering Institute, 2002, http://www.sei.cmu.edu/cmmi/ [Last Accessed April 2005].

Gerstner, Louis V. Jr., *Who says Elephants Can't Dance?* HarperCollins, 2002. ISBN 0007153538.

Gilb, Kai, *Evo*, 2005. Draft manuscript at http://www.gilb.com

Gilb, Tom and Graham, Dorothy, *Software Inspection*, Addison Wesley, 1993.

Gilb, Tom, "Software Project Management: Adding Stakeholder Metrics to Agile Projects", *Novática*, Issue 164, July-August 2003. (Special Edition on Software Engineering - State of an Art, Guest edited by Luis Fernández-Sanz. Novática is the journal of the Spanish CEPIS society ATI (Asociación de Técnicos de Informática.)
See http://www.upgrade-cepis.org/issues/2003/4/upgrade-vIV-4.html
In Spanish: http://www.ati.es/novatica/2003/164/nv164sum.html

Gilb, Tom, "Adding Stakeholder Metrics to Agile Projects", *Cutter IT Journal: The Journal of Information Technology Management*, July 2004, Vol. 17, No.7, pp31-35. See http://www.cutter.com.

Gilb, Tom, *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering using Planguage*, 2005, Elsevier Butterworth-Heinemann. ISBN 0750665076.

Larman, Craig and Basili, Victor R., "Iterative and Incremental Development: A Brief History", *IEEE Computer Society*, June 2003, pp 2-11.

Larman, Craig, *Agile and Iterative Development: A Manager's Guide*, Addison Wesley, 2003. See Chapter 10 on Evo.

Morris, Peter W. G., *The Management of Projects*. London: Thomas Telford. 1998. ISBN 072771693X. 358 pages. USA: The American Society of Civil Engineers.

Neill, Colin J., and Laplante, Phillip A., "Requirements Engineering: The State of the Practice", *IEEE Software*, November/December 2003, pp. 40-45.

Agile Delivery at British Telecom

Ian Evans, ian.l.evans@bt.com
British Telecom, www.bt.com

Introduction

It is becoming clear, not least from the pages of this publication, that agile development methods are being adopted or at least considered by a growing number of software development teams & organisations. Whether you are already an active practitioner agile development, or considering its adoption on your project, you will be aware of the business benefits that can be derived through faster and more effective software delivery not to mention the motivational impact it can have on development teams. Alternatively, maybe you work for a large organisation that has yet to make any serious inroads into agile development, and are left wondering how agility could be made to work on a large scale.

If you're in the latter camp, or even if you are not actively considering agile development as such but are struggling to deliver large and / or complex programmes using traditional approaches and wishing there was a better way, then you are probably where British Telecom (BT) found itself in 2004. That was before the arrival at the company of a new CIO who systematically set about replacing the company's long-standing waterfall-based delivery processes with one that embodied the key principles of agile delivery.

This article presents an overview of the approach taken by BT, illustrating how agile development principles can be applied successfully at the enterprise level. Needless to say, the approach taken by BT is not for the faint hearted – it has included a high degree of risk, and certainly a lot of pain. Now well into its second year however, although the transformation is far from complete, it is already paying dividends.

Background

BT employs some 8,000 IT professionals in a variety of roles including project & delivery management, architecture & design, software engineering, integration & testing, operational support and service management. Much of its internally-focussed development work has traditionally been channelled through a number of business-focussed delivery projects or programmes, ranging from quite small, simple developments to large-scale and complex business solutions, the latter tending to be the norm.

The predominant delivery approach, certainly for the larger delivery programmes, was very much waterfall-based. The use of agile development practice, notably DSDM and Scrum, was limited to a small number of fairly small, self-contained development teams. BT was in fact one of the founding members of the DSDM Consortium and took an active part in shaping the method in its early days.

Despite successfully delivering a number of large, complex solutions into a dynamic, competitive yet highly regulated business environment, many significant transformation programmes were struggling to deliver any notable results in an acceptable timeframe. As part of a CMMI-inspired improvement strategy, efforts had been made to formalise acknowledged best practice processes into a standard delivery methodology. In 2004, this standard methodology was in the process of being rolled out when the new CIO made it clear that an entirely new agile approach was needed.

Drawbacks of the waterfall

Reinforcement of current waterfall-based practices was not really the answer however. Many of the delivery problems experienced at BT, and no doubt other large organisations, stem from the nature of the waterfall lifecycle itself. Some examples of these problems are given here. For a more complete demolition of waterfall practices, refer to Craig Larman's excellent work [1].

Poor requirements capture

Capturing requirements certainly isn't a bad thing. On typical large programmes however,

- Individual business stakeholders are anxious to incorporate all of their known requirements into the first / next release
- "Gold users" generate hundreds, if not thousands of detailed requirements that often bear little relationship to the business problems that needs to be addressed
- Most if not all requirements are given a high priority
- The requirements themselves, at best, represent today's view, which will certainly have changed by the time the requirements are actually implemented

Disconnected design

Given the sheer number of requirements, the design community finds itself spending most if its time trying to figure out what they mean. Meanwhile,

- The requirements analysts move on to other projects, taking with them important tacit knowledge
- Some stakeholders become concerned that their requirements are not being adequately addressed, and therefore refuse to sign off the designs
- Other stakeholders unearth more requirements or raise change requests, diverting scarce design expertise onto impact analyses

Development squeeze

With the design stage having slipped, development teams find themselves under intense pressure to deliver components into the integration environment by the originally agreed date. In fact, they often take the decision, reluctantly, to start development against an unstable design, rather than do nothing or divert resources to other programmes. Inevitably, system testing is cut short so that original timescales are met and the programme is seen to be on target.

The integration headache

The integration team has a set number of weeks during which it needs to integrate what it expects to be fully functional and relatively bug-free code. Because of the instability of the component code, and the lack of any effective regression test capability, effort is instead diverted to trying to resolve elementary bugs in the delivered code, liaising with a development team that is now engaged in the next major release. Actual integration therefore runs into months, creating a knock-on effect on other programmes requiring the services of the Integration team, not to mention frustrations within the business community who had been busy preparing themselves for an on-time delivery.

The deployment nightmare

It is now at least 6, or even 12 – 18 months since the business originally identified the need for this particular solution. Compromises and oversights made during the requirements and design phases, followed by de-scoping during development has resulted in a solution that bears little relationship with what was originally envisaged. Besides, the world has actually moved on in the meantime. The business then finds that the solution is not fit-for-purpose and refuses to adopt it. Worse, they adopt it and soon find that it is slow, error-prone and lacks key features, and eventually revert to the old system. The end result – more shelfware!

Agile Development

Although not intended to be a “silver bullet”, Agile Development should help resolve these problems. For example,

- Focussing only on what’s really important to the business will help overcome the tendency to want to document all requirements up-front. Techniques such as User Stories [2] often proves helpful here.

- Involving your customer as part of your team certainly helps ensure that the solution is developed in line with expectations, and doesn't incur too many surprises once completed
- Developing in short iterative cycles again helps to break the problem down into more manageable chunks, and allows feedback to be gleaned early and often. The Scrum [3] method, for instance, advocates 30-day cycles.
- Applying a 'test-first' and continuous integration development approach, two of the core practices of XP [4], not only helps to further break down the complexity of the problem but also avoids the 'integration headache' described above

The Challenges of Enterprise Agile

This is all very well when you have a development team of some 10 or 20 people, or if you are implementing agile development on perhaps one of your larger (say, 100 people) projects and have ample mentoring resources at your disposal. Even then, if you're adopting Agile for the first time, you have the challenge of changing people's mindsets, aligning the new practices with a supportive customer, while also remembering to successfully deliver the actual project.

For most large organisations, architectural considerations also need to be addressed. Also, no doubt a high proportion of the code base is probably implemented in the form of third-party Commercial Off-The Shelf (COTS) components. On top of this, you then have the added complexities of off-shore development, widely distributed in-house resource, an IT department that is probably not highly regarded within the business, plus a whole pile of legacy code for which tests probably don't exist in any shape or form let alone automated.

Finally, your programme has significant business commitments to make, and little scope to take on the added risk of adopting new practices. So where do you start?

Scaling to the Enterprise – The BT Approach

At BT, the drive towards agile delivery started with an uncompromising determination to introduce shorter delivery cycles across the entire delivery programme portfolio, establish a ruthless focus on delivering real business and end-customer value, and to nurture a strong collaborative ethos between the IT and Business communities.

90-day cycles

With more or less immediate effect, all delivery programmes were required to adopt a standard 90-day delivery cycle. That is, having picked up one or more distinct business problems on day 1, a working solution is expected to be available, fully-tested, for deployment by the end of the remaining 90 calendar days. For BT, this represented a seismic shift from the 12+ month delivery cycles that were previously commonplace.

Each 90-day cycle is kick-started by an intensive 3-day "hot house" in which cross-functional teams explore one or more business problems in some detail, with the main stakeholder(s) usually being at hand to resolve any queries. Out of each hot house, the intention is that one or more working prototypes are produced which, if accepted by the stakeholder, forms the basis of the development activity for the remainder of the cycle.

For the remainder of each cycle, the intention is that wherever practical, programmes pursue agile development practices such as more fine-grained iterative development (e.g. 2 – 4 weeks), test-driven development, and continuous integration.

Step 2 – Focus on Delivering Business Value

Early in each delivery cycle, the programme sets out clear targets for what it expects to achieve for the business during that cycle. These targets invariably include a strong emphasis on the end-customer experience, such as overall response times, transaction success rates, and so on. At the end of the cycle, the programme is assessed against these targets, and the outcome of this assessment will influence the timing of bonus payments for the programme team members. Programmes failing to deliver business value over a series of cycles face being closed down altogether.

This of course places a certain amount of pressure on the (internal) customer to be clear about the business priorities and the features that would provide the greatest return on investment. It also requires that the customer is ready and able to deploy the solutions into the business and realise the intended benefits. In practice, programmes often take two or more 90-day cycles to progress a particular solution to a point where it is fit for deployment. Even so, there is an opportunity at the end of each cycle to assess what has been delivered so far, and to provide feedback based on what has already been developed.

Step 3 – Instil a Collaborative approach

Truly successful programmes require a strong partnership approach between the business customer and the development community. Within BT, close collaboration is established at the outset of each project through attendance at the hot houses. The onus is then on the programme teams to ensure this collaboration continues throughout the delivery cycle through design walkthrough sessions, prototype reviews, and so on.

In practice, the end-user representatives you want to have at your hot house are also the ones who are hardest to release from operational duties. With several programmes running numerous hot houses during the course of a year, this problem can quickly become compounded and often makes it extremely difficult to achieve true collaboration on a day-to-day basis. However, collective ownership of the eventual solution needs to become the accepted norm, and any practical steps to enhance collaboration should be taken.

Early Reflections

Despite some turmoil at the start, and some painful failures among some of the earlier hot houses & delivery cycles, the new practices have now become accepted as the norm across BT. Now well into the second year of its shift from waterfall to agile delivery practices, few people would be willing to revert to pre-Agile practices. In fact, most programmes are now seeking ways of refining their delivery processes further by adopting truly iterative & test-driven development practices within each delivery cycle.

However, some observations would be worth noting.

- Firstly, when you're embarking on an agile delivery strategy at the enterprise level, it is imperative to quickly establish a 'critical mass' of people who not only grasp the ideas behind it but are also comfortable with its application. To establish that critical mass, you will probably need to turn to outside help. A number of consultancies now specialise in the adoption of agile practices within large organisations. BT chose to use two different companies, each of which brought different strengths and perspectives. Further to this, it is also essential to establish a strong central team to provide ad-hoc support, nurture the new techniques, and to actively support the new practices.

- Certain agile practices, such as test-driven development, are harder to adopt when most of your development is based on legacy code and / or externally-sourced components. Similarly, continuous integration becomes extremely complex when some of you main components are shared across multiple programmes. Some of BT's programmes are now pursuing test-first and continuous integration techniques, but this takes time and investment and is only being done on a selective basis.
- For Agile Development to work at the enterprise level, you still need to pay due attention to your systems architecture. "Big Design Up-Front" (BDUF) may not appeal to the agile purist, but re-factoring of an enterprise architecture simply isn't practical.
- Not all delivery activity fits neatly into the agile development model. Given a choice however, the natural tendency is to pursue most activities using the traditional approaches – you can always find some excuse why "the new approach" isn't appropriate on your project. If you go down this road, agile delivery will at best become a niche activity. At BT, a strong mandate ensured that all programmes put the new practices to the test whether this seemed logical or not. This helped to break through the "pain barrier" and to ensure that the new practices were given a real chance of taking hold.
- To be truly effective, the agile approach needs to reach right across the business, not just the IT organisation. You might expect that the business would be excited at the prospect of having regular deliveries of valuable functionality. However, the business also needs to move away from traditional waterfall practices and change how it engages with the IT organisation. It also has to place its trust in the IT organisation (something that certainly takes time) that it will deliver as promised. It then needs to ensure that it is geared up to exploit the deliveries to gain maximum business benefit.
- Finally, remember the old adage – "There's no gain without pain!" Applying the principles described here on large projects or programmes in typical large organisations requires courage, determination, and no small degree of risk. Also, such a radical strategy requires absolute commitment from the very top.

Conclusion

Re-orienting a large IT organisation from pursuing well-established waterfall-based delivery approach to being a truly agile delivery unit takes patience and time, as well as a lot of commitment. In BT, where the initial steps towards enterprise agile delivery were taken late 2004, there has been a noticeable and decisive shift away from waterfall-based thinking. It has also transformed, quite radically, the traditional function of the IT department as a supplier of IT services to one where IT is now seen as integral to all major business initiatives. Above all else, it has created an attitude, bordering on obsession, of delivering real value to the business through IT.

Despite the early successes however, it is clear within BT that there is still a long way to go before it can consider itself to be truly agile. For any large organisation, the journey from waterfall to agile can be very long and challenging. As with other proponents of Agile Development however, few at BT would want to turn back to the old ways.

References

1. "Agile & Iterative Development" by Craig Larman, Addison-Wesley (2004) ISBN: 0-13-111155-8
2. "Agile Software Development with Scrum" by Ken Schwaber & Mike Beedle, Prentice Hall (2002) ISBN: 0-13-067634-9
3. "User Stories Applied for Agile Software Development" by Mike Cohn, Addison Wesley (2004) ISBN: 0321205685
4. "Extreme Programming Explained" by Kent Beck & Cynthia Andres, Addison Wesley (2004) ISBN: 0321278658
5. "Lean Software Development – An Agile Toolkit" by Mary Poppendieck & Tom Poppendieck, Addison Wesley (2003) ISBN: 0321150783
6. Agile Manifesto – http://www.agilemanifesto.org

Running an Open Source Software Project

William Echlin, QaTraq Project Manager,
www.testmanagement.com

Some people would be happy to convince you that managing an Open Source Software (OSS) project is completely different than managing a commercial software project. People working on Open Source software argue that there are no deadlines to meet, that quality issues can be left for the community of users to identify, and that their are no complications of costing and budgeting to manage. I will hopefully have convinced you of the contrary by the end of this article. I will have showed you that managing a normal software project and an open source project has far more parallels than most people would have you believe.

Do the common project life cycle principals apply to the average Open Source Software project? Do OSS projects go through the phases of definition, planning, organising, execution and closure? It is perhaps not quite as formal, but if you look hard enough the same phases of the development process can be found in all OSS projects. Whether it's conscious or subconscious, OSS projects all follow the common development stages. In fact it can probably be argued that the successful OSS projects follow these principles intuitively and instinctively. Good project management practices really can make the difference between successful and unsuccessful OSS projects.

Take for example the appointment of a project manager. The majority of OSS projects don't formally appoint an official project manger, yet in most projects you will find someone taking on the role of project manager. He understands the significance of the development process and the importance of leading a good team.

So what is it that an unofficial OSS project manager might bring to the successful OSS project? As mentioned earlier, the project manager can't ignore the familiar stages making up the common software development process:

1. Project Definition
2. Planning
3. Organising
4. Executing
5. Closing

However, following this development process amounts to very little if the project manager has little understanding of the key leadership skills involved. The ability to specify precise goals, to communicate clearly and to motivate members of the project is crucial to the leadership. In the Open Source environment, the motivation of the team members presents the real challenge. The familiar financial motivation plays no part in many open source projects. For most developers working in an OSS environment, you will probably find the following motivators having the biggest impact on the project:

- Achievement
- Recognition
- Responsibility
- Advancement

None of these motivators are tangible, but you have to pay close attention to them in OSS projects. It is easy in commercial projects to over look these points and focus on the more tangible motivator known as money. If you want a truly motivated team though, concentrate on both the tangible and the intangible motivators!

Even in the apparently chaotic world of open source software development, clear process and good leadership are essential tenets. The process may be more fluid in an OSS project and the leadership less formally defined, but both aspects are just as important all the same. In the following sections we will look at the stages of the project development process in the OSS context and see how team motivation plays its part in each of these stages.

Project Definition

"Project Definition" sounds very formal, but its importance can't be too highly stressed. The problem definition influences everything within your project. Whilst working on an OSS project we may not formally document that we need achieve X and Y, but in many cases the objectives of an OSS project are far clearer in the minds of the OSS project team than that of the commercial software development team.

In the case of a commercial software development team, you might typically have someone define what needs to be achieved and then communicate that vision to the development team, hoping that they understand it. In the OSS scenario, you typically have a group of developers that have come together because they have all experienced, first hand, the "problem definition". This makes it is far easier to understand the problem and see a route to the solution.

Defining the 'problem' and specifying the requirements isn't always straightforward. However, this is where most OSS projects have an advantage over their commercial counterparts. By far the best way to understand a requirement is not to read it, but to experience it. Most OSS developers have experienced the need to fulfil a particular requirement. After all, coming up against the problem is probably the main reason they've picked a particular OSS project to work on in the first place. Experiencing the need for a requirement first hand will always provide a better understanding than reading a written requirement specification second hand.

With a clear definition of the problem you are already ahead when it comes to motivating the team. Having a clear goal is absolutely key to engendering a feeling of achievement as the project evolves. You can only feel like you've achieved something, or that you are progressing, if you know what you are aiming for. So make sure you define the project and that you communicate that definition clearly to the rest of the team.

Planning

When you think of Open Source projects, formal project planning isn't one of the first tasks that spring to mind. Open Source projects have a reputation for a slightly more ad-hoc approach to planning. Maybe the OSS approach to planning isn't formal but, believe it or not, it is still a step that needs to be taken seriously. The OSS projects that succeed may not have formally defined project plans, but you can guarantee that the team has an instinctive ability to plan and organise their work.

The almost religious reliance on defect tracking tools (e.g. Bugzilla or Mantis) is possibly one of the reasons OSS teams are so good at identifying and organising tasks. In the OSS environment, the teams defect-tracking tool turns into far more than just a tool for tracking defects. It becomes the foundation that helps to organise and priorities the work of the whole team. It tracks everything from release tasks, defects, enhancement suggestions and, sometimes, even the to-do lists of individual developers. The defect tracking system's effectiveness at prioritising and assigning tasks comes into its own within many OSS projects. Without the pressures of deadlines, the complexity of tools like Microsoft Project can be left behind for defect tracking tools that are far easier to implement and run.

The use of system architecture and design definition is possibly the weakest link in this comparison with formal project management. Whilst in commercial projects it is common to see reams of paper specifying the design of the system, it is not common to see this sort of work on an OSS project. In a typical OSS project, it is common for design ideas and concepts to be quickly prototyped in code and distributed to the community for feedback and comment. Even if it is perhaps not the most efficient use of time and resources, this proves to be an effective mechanism for identifying what should and should not be included in the application. Personally I advocate a least a certain degree of formal design work before coding begins. Formal design specifications add clarity to the project and help foster a feeling of advancement as the project progresses.

It is during the planning phase that serious consideration needs to be given to matching the goals of the project with the goals of the team members. One of the key reasons people get involved in OSS projects is to improve their skills and experience. If you decided to implement your project using Pascal, you would likely limit the pool of potential developers to work on the project, or even empty it. If, however, you select one of the more popular and exciting technologies, you are more likely to attract and retain coders on your project. Again the feeling of advancement whilst building and developing new skills helps to create an environment of motivation.

Organising

The OSS team usually excels at this stage. The ability to bring together the team members, the tools, controls and communications methods to get the job done are second nature to most OSS teams (partly because they know exactly where to turn to for OSS solutions to address these issues and partly because that are no organisational restrictions imposed on the implementation). A typical OSS team thinks nothing of implementing the tools needed to run the project efficiently. Setting up a defect tracking system, a forum and source code control in days if not hours.

The difficult, and perhaps crucial part, is how you open up these tools to the community. Do you open up your defect tracking system to absolutely everybody? Thereby exposing yourself to perhaps hundreds of poorly written, invalid defect records which all need sorting through. Do you provide easy access for new members of the development team to the source code repository? People who have no track record on the project could make critical changes to the code base.

I witnessed a recent exchange on an OSS project forum where a new member had been busy checking in code changes to the CVS repository. He had renamed fundamental aspects of the application because he thought he knew better about the terminology that should be used. This demonstrated how easily extra, unnecessary work can be created if you don’t get the project controls right. It is difficult to get it right as to whom, how, when and where you open up your source code repository. Yet getting it right is essential to the success of the project.

It is essential to get the balance right between restrictive controls and the freedom that helps motivate the team. If your controls are draconian you stifle enthusiasm and motivation. If you loosen controls you may find it easier to develop the levels of motivation. Giving your team more responsibility makes a big difference to the project and can be incredibly motivating. There is nothing complicated with principals, but never underestimate the importance of getting the balance right for your project.

It comes down to making the right decisions in involving the community you are serving and keeping the control and direction of the project on the right path. Like many things in life, the solution usually lies somewhere between the two extremes and can depend largely on the maturity of the project. Never forget though, that passing on more responsibility can be a powerful motivator.

Executing

You would think that the coding stage would be a walk in the park for most OSS projects. After all, we're all supposed to be good at this the part. Yet the success of this part of the project is largely dependent on the foundations built in the previous stages of the project. If you don't have a clear understanding of the problem you are solving or your prioritisation of the work was short of the mark, you limit the chances of success. Good coding alone doesn't make a successful project.

Assuming that the foundation stages have gone well, it is this stage where the “release early and often” approach is often cited as being the key to a successful OSS project. However, I would argue that an OSS project that relies solely on the community for its unit and system testing is asking for trouble. I recently upgraded to the latest version of a popular Linux operating system. Maybe it was my fault for not reading the release notes properly, but by the time I realised that they’d stop supporting a popular database that I relied on it was too late. Yes they released early, but I had always found previous early releases reliable and became complacent when upgrading. This taught me a valuable lesson regarding complacency and the deployment of OSS.

There is a balance to get right here, especially now that more and more people are starting to rely on OSS. If you continually release buggy software in today’s environment, you risk losing users. You can't expect users to test everything for you. With so much choice around now (I’ve lost count of the number of Linux distributions now), users will remain loyal only if you reach a certain level of quality before release. If you abuse the loyalty of those users, then you’ll have fewer users to further support your testing efforts. With fewer users you have less testers and you enter a slow but lethal spiral of death. Get it right though and you can expect a faithful, loyal following of users.

Feedback from users is another absolutely crucial aspect to the execute stage. Few OSS projects enjoy the privileges of a dedicated test team that is paid to give you feedback. This presents two key problems

1. the test team / users won’t be physically located near you
2. the users / testers aren't obliged to give you feedback.

To a large extent the feedback you get is down to two things

1. how easy you make it for people to provide feedback
2. how supportive you are to those people when they provide feedback.

This bit isn't difficult to understand. If they can't provide you with feedback (i.e. you don't give them a usable feedback channel) you won't get any feedback. If you don't thank your users/testers or you aren't grateful, then they won't provide you with feedback a second time round. More than commercial projects, OSS projects live and die by the feedback they receive from users, because they have no internal feedback mechanisms or internal test team to rely on.

Forums are among the best feedback channels and the most powerful motivators for OSS projects. Forums are so powerful that I find it difficult to understand why commercial projects don't use of them more. Perhaps it’s the thought of airing your dirty washing in a forum that puts commercial projects off using forums. Yet, if a developer in a commercial project receives negative feedback about his/her work in a forum, you can almost guarantee that he/she will feel motivated to do something positive about it. We all crave for feedback and recognition. Feedback through forums can satisfy those cravings.

Take, as an example, the last time you ate at a restaurant and complimented the waiter for a really delicious meal. The chef cooked the meal but we compliment the waiter. How do we know that the compliment was sent back to the chef? Wouldn’t it be more rewarding for the chef if he was complimented in person? It’s no different in software development, as forums can provide that direct channel between the users and developers. A forum is like standing up in the restaurant after finishing your meal and shouting through to the chef in the kitchen that you thought the meal was delicious. Not only does the chef receive your complement directly, but you’ve also told the rest of the dinners in the restaurant what you thought about the meal. That’s a huge motivator for the chef.

Closing

Of all the stages, this is the one that is very different to that of a commercial project. Getting it right can make a huge difference to the overall image of the project. You don't have to provide documentation. You don't have to provide any support. You don't have to make sure all defect fixes are verified before release. In a typical commercial project, you will not achieve sign off until all of these aspects are complete. Yet many people fail to realise that if you don't complete these aspects with OSS projects, you are forcing your users to jump though many unnecessary hoops, which is likely to mean less users.

The Apache foundation is a really good example of completing the documentation successfully. People actually come forward to volunteer to write documentation for this project, because they receive kudos and recognition for their efforts. In smaller and less successful projects, this can be however an extremely difficult step to complete. Nonetheless, without good documentation it is very difficult to call an OSS project a success. Users are so dependent on well-written documentation. Imagine trying to use CVS without a well written user guide... it wouldn't be impossible but it would be far more difficult.

It is also during this phase that the feeling of achievement comes from knowing you've solved the problem defined at the start of the project. This is why it is so important to define the problem in the first place. When members of the team know they've solved the problem defined at the start, they know they've been part of a successful project. It's important to remember that a successful project is defined as much by the members of the team working on the project as it is by the end users.

Conclusion

Recognition, responsibility, advancement and a feeling of achievement all play a crucial part in keeping a team motivated. It is that motivation that plays one of the biggest parts in making a project a success. No amount of project definition, planning and organising will achieve anything if your team isn't motivated. Yet every stage of a project presents opportunities for you to motivate your team. Whether you are running an Open Source project or a commercial, closed source, project, you should take every single one of these opportunities to motivate you team because this will ultimately create successful projects.

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