Cursus

Software Life Cycle

Reader

Open Universiteit
Faculteit Management, Science & Technology

Cursusteam
Dr. ir. K.A.M. Lemmen, *cursusverantwoordelijke en auteur*
Ir. P. Oord, *auteur*

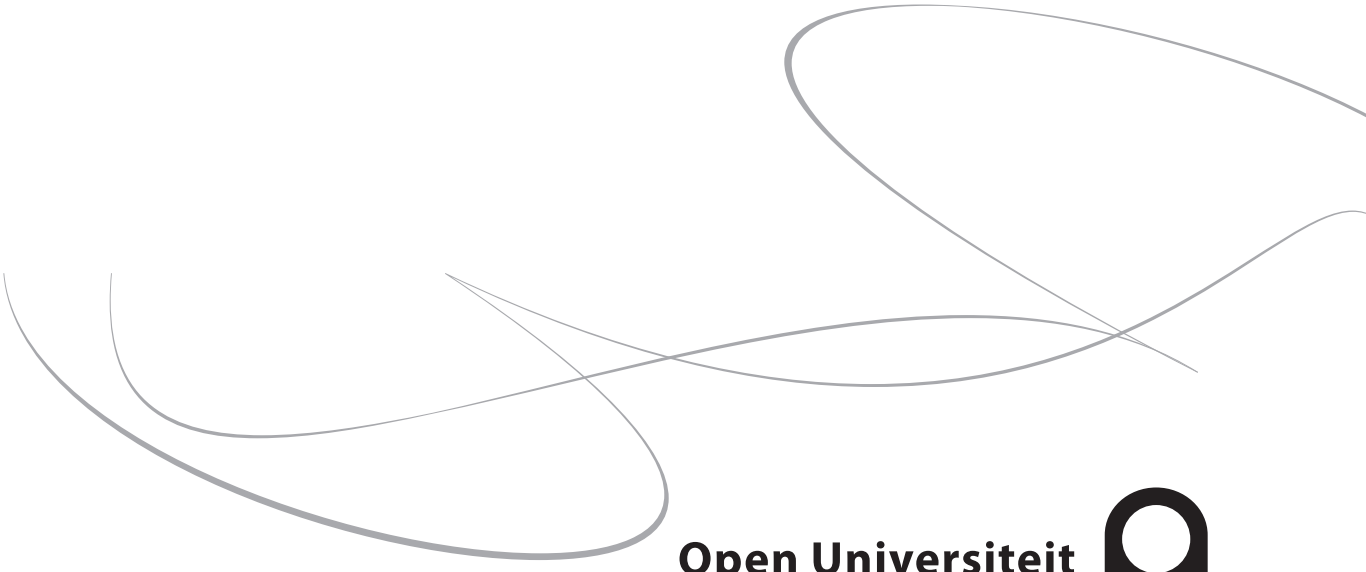
Deze cursus is grotendeels samengesteld uit delen van de cursussen Softwaremanagement (T24331) en Requirements engineering (T65311), waaraan de volgende auteurs hebben meegewerkt:
drs. M.J. te Boekhorst
ir. P. Oord
drs. Th.F. de Ridder
drs. H.J. Sint
ir. S. Stuurman
dr. W. Westera

Extern referenten
Prof. dr. A. van Lamsweerde
Dr. ir. D.M. van Solingen

Programmaleiding
Prof. dr. M.C.J.D. van Eekelen

READER

Software Life Cycle



Open Universiteit
www.ou.nl

Productie
Open Universiteit

Redactie
John Arkenbout

Lay-out en illustraties
Maria Wienbröker-Kampermann

Omslag
Team Visuele communicatie, Open Universiteit

Druk– en bindwerk
OCÉ Business Services

Eerste druk: 2015

IM0303_50108_21012015

ISBN 978 94 91825 84 2 (reader)
ISBN 978 94 91825 92 7 (serie)

Cursuscode IM0303

Onderdeel	Leereenheid	Studielast	Bladzijde
Werkboek	Introductie tot de cursus		
	1 Life Cycle Models	10	
	2 Projectmanagement	10	
	3 Agile Life Cycles	10	
	4 Process Improvement	10	
	5 Softwarekwaliteit	10	
	6 Requirements Engineering: Setting the Scene	15	
	7 Domain Understanding and Requirements Elicitation	8	
	8 Goal Orientation in Requirements Engineering	7	
	9 Modelling System Objectives with Goal Diagrams	10	
	10 Risk Analysis on Goal Models	8	
	11 A Goal-Oriented Model-Building Method in Action	6	
Reader	Artikelen bij leereenheid 1		
	1 A Spiral Model of Software Development and Enhancement		7
	2 Bridging Agile and Traditional Development Methods: A Project Management Perspective		33
	3 Improving Software Economics		39
	Artikelen bij leereenheid 2		
	4 The Mythical Man-Month, after 20 Years		81
	5 The Profession of IT, Evolutionary System Development		87
	6 In-House Software Development: What Project Management Practices Lead to Success?		91
	Artikelen bij leereenheid 3		
	7 De ScrumgidsTM		101
	8 Scrum Development Process		119
	9 Scrum Anti-patterns – An Empirical Study		139
	Artikelen bij leereenheid 4		
	10 Process Improvement for Small Organizations		149
	11 Successful Process Implementation		157
	12 Measuring the ROI of Software Process Improvement		167
	13 A tale of two cultures		175
	Artikelen bij leereenheid 5		
	14 Softwarekwaliteit, wat is dat?		195
	15 Het realiseren, handhaven en garanderen van een goed softwareproces		215
	16 Invloed van de factor mens op softwarekwaliteit		223
	17 The Darker Side of Metrics		243
Tekstboek	Requirements Engineering, From System Goals to UML Models to Software Specifications, Axel van Lamsweerde, Wiley, 2009		
Cursusweb	http://studienet.ou.nl		
	Nieuws, begeleiding, aanvullingen, discussiegroep		

A Spiral Model of Software Development and Enhancement
Barry W. Boehm

ACM SIGSOFT Software Engineering Notes, volume 11, issue 4, August 1986

A Spiral Model of Software Development and Enhancement

Barry W. Boehm, TRW Defense Systems Group

"Stop the life cycle—I want to get off!"
"Life-cycle Concept Considered Harmful."
" The waterfall model is dead."
"No, it isn't, but it should be."

These statements exemplify the current debate about software life-cycle process models. The topic has recently received a great deal of attention.

The Defense Science Board Task Force Report on Military Software[1] issued in 1987 highlighted the concern that traditional software process models were discouraging more effective approaches to software development such as prototyping and software reuse. The Computer Society has sponsored tutorials and workshops on software process models that have helped clarify many of the issues and stimulated advances in the field (see "Further Reading").

The spiral model presented in this article is one candidate for improving the software process model situation. The major distinguishing feature of the spiral model is that it creates a *risk-driven* approach to the software process rather than a primarily *document-driven* or *code-driven* process. It incorporates many of the strengths of other models and resolves many of their difficulties.

This article opens with a short description of software process models and the issues they address. Subsequent sections outline the process steps involved in the spiral model; illustrate the application of the spiral model to a software project, using the TRW Software Productivity Project as an example; summarize the primary advantages and implications involved in using the spiral model and the primary difficulties in using it at its current incomplete level of elaboration; and present resulting conclusions.

Background on software process models

The primary functions of a software process model are to determine the *order of the stages* involved in software development and evolution and to establish the *transition criteria* for progressing from one stage to the next. These include completion criteria for the current stage plus choice criteria and entrance criteria for the next stage. Thus, a process model addresses the following software project questions:

(1) What shall we do next?

(2) How long shall we continue to do it'?

Consequently, a process model differs from a software method (often called a methodology) in that a method's primary focus is on how to navigate through each phase (determining data, control, or "uses" hierarchies; partitioning functions; allocating requirements) and how to represent phase products (structure charts; stimulus—response threads; state transition diagrams).

Why are software process models important? Primarily because they provide guidance on the order (phases, increments, prototypes, validation tasks, etc.) in which a project should carry out its major tasks. Many software projects, as the next section shows, have come to grief because they pursued their various development and evolution phases in the wrong order.

Evolution of process models. Before concentrating in depth on the spiral model, we should take a look at a number of others: the code-and-fix model, the stage-wise model, the waterfall model, the evolutionary development model, and the transform model.

The code-and-fix model. The basic model used in the earliest days of software development contained two steps:

(1) Write some code.

(2) Fix the problems in the code.

Thus, the order of the steps was to do some coding first and to think about the requirements, design, test, and maintenance later. This model has three primary difficulties:

(a) After a number of fixes, the code became so poorly structured that subsequent fixes were very expensive. This underscored the need for a design phase prior to coding.

(b) Frequently, even well-designed software was such a poor match to users' needs that it was either rejected outright or expensively redeveloped. This made the need for a requirements phase prior to design evident.

(c) Code was expensive to fix because of poor preparation for testing and modification. This made it clear that explicit recognition of these phases, as well as test and evolution planning and preparation tasks in the early phases, were needed.

The stagewise and waterfall models. As early as 1956, experience on large software systems such as the Semi-Automated Ground Environment (SAGE) had led to the recognition of these problems and to the development of a stagewise model[2] to address them. This model stipulated that software be developed in successive stages (operational plan, operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, and system evaluation).

The waterfall model,[3] illustrated in Figure 1, was a highly influential 1970 refinement of the stagewise model. It provided two primary enhancements to the stagewise model:

(1) Recognition of the feedback loops between stages, and a guideline to confine the feedback loops to successive stages to minimize the expensive rework involved in feedback across many stages.

(2) An initial incorporation of prototyping in the software life cycle, via a "build it twice" step running in parallel with requirements analysis and design.

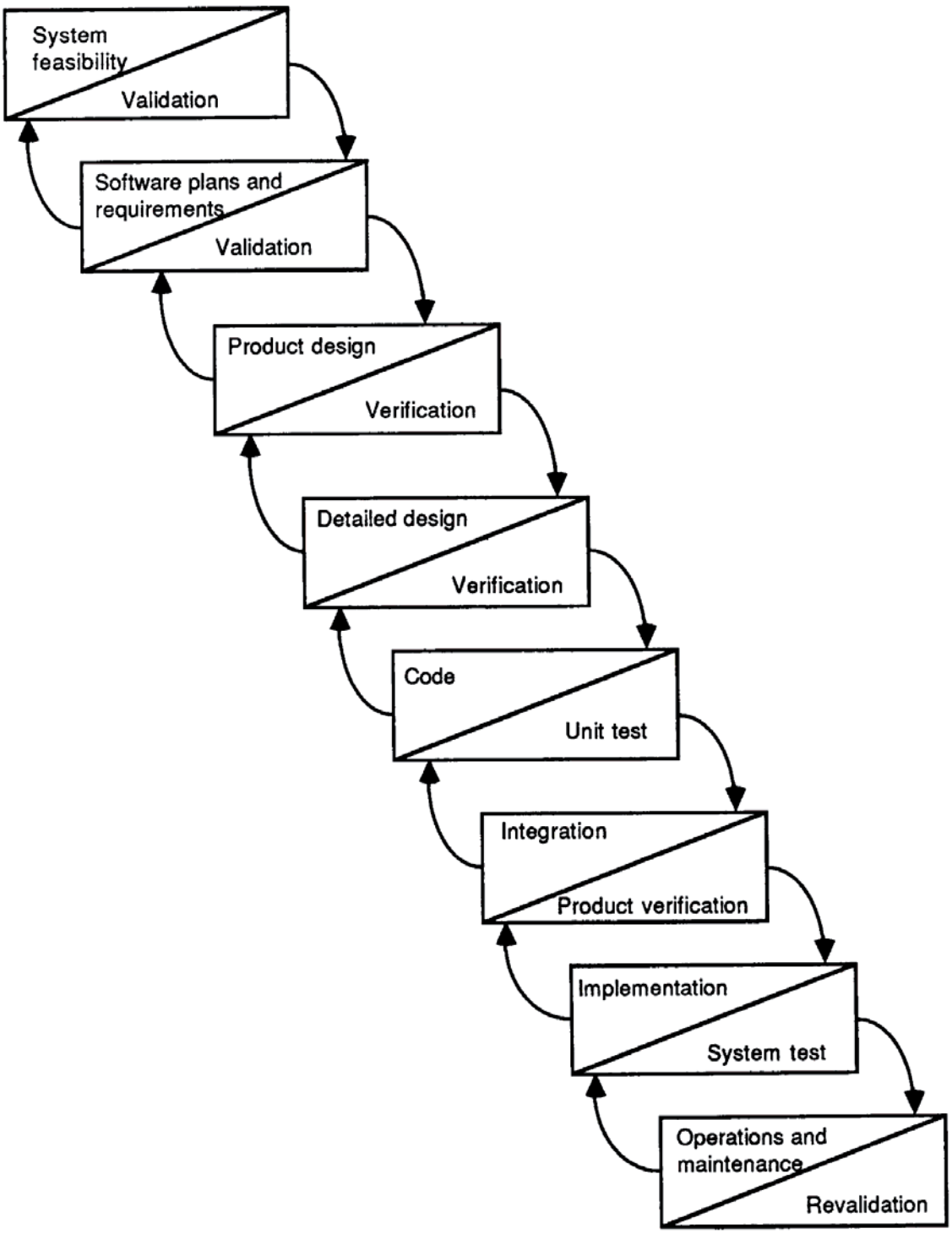


Figure 1. The waterfall model of the software life cycle.

The waterfall model's approach helped eliminate many difficulties previously encountered on software projects. The waterfall model has become the basis for most software acquisition standards in government and industry. Some of its initial difficulties have been addressed by adding extensions to cover incremental development, parallel

developments, program families, accommodation of evolutionary changes, formal software development and verification, and stagewise validation and risk analysis.

However, even with extensive revisions and refinements, the waterfall model's basic scheme has encountered some more fundamental difficulties, and these have led to the formulation of alternative process models.

A primary source of difficulty with the waterfall model has been its emphasis on fully elaborated documents as completion criteria for early requirements and design phases. For some classes of software, such as compilers or secure operating systems, this is the most effective way to proceed. However, it does not work well for many classes of software, particularly interactive end-user applications. Document-driven standards have pushed many projects to write elaborate specifications of poorly understood user interfaces and decision support functions, followed by the design and development of large quantities of unusable code.

These projects are examples of how waterfall-model projects have come to grief by pursuing stages in the wrong order. Furthermore, in areas supported by fourth-generation languages (spreadsheet or small business applications), it is clearly unnecessary to write elaborate specifications for one's application before implementing it.

The evolutionary development model. The above concerns led to the formulation of the *evolutionary development* model,[4] whose stages consist of expanding increments of an operational software product, with the directions of evolution being determined by operational experience.

The evolutionary development model is ideally matched to a fourth-generation language application and well matched to situations in which users say, "I can't tell you what I want, but I'll know it when I see it." It gives users a rapid initial operational capability and provides a realistic operational basis for determining subsequent product improvements.

Nonetheless, evolutionary development also has its difficulties. It is generally difficult to distinguish it from the old code-and-fix model, whose spaghetti code and lack of plan-fling were the initial motivation for the waterfall model. It is also based on the often-unrealistic assumption that the user's operational system will be flexible enough to accommodate unplanned evolution paths. This assumption is unjustified in three primary circumstances:

(1) Circumstances in which several independently evolved applications must subsequently be closely integrated.

(2) "Information-sclerosis" cases, in which temporary workarounds for software deficiencies increasingly solidify into unchangeable constraints on evolution. The following comment is a typical example: "It's nice that you could change those equip-ment codes to make them more intelligible for us, but the Codes Committee just met and established the current codes as company standards."

(3) Bridging situations, in which the new software is incrementally replacing a large existing system. If the existing system is poorly modularized, it is difficult to provide a good sequence of "bridges" between the old software and the expanding increments of new software.

Under such conditions, evolutionary development projects have come to grief by pursuing stages in the wrong order: evolving a lot of hard-to-change code before addressing long-range architectural and usage considerations.

The transform model. The "spaghetti code" difficulties of the evolutionary development and code-and-fix models can also become a difficulty in various classes of waterfall-model applications, in which code is optimized for performance and becomes increasingly hard to modify. The transform model[5] has been proposed as a solution to this dilemma.

The transform model assumes the existence of a capability to automatically convert a formal specification of a software product into a program satisfying the specification. The steps then prescribed by the transform model are

- a formal specification of the best initial understanding of the desired product;
- automatic transformation of the specification into code;
- an iterative loop, if necessary, to improve the performance of the resulting code by giving optimization guidance to the transformation system;
- exercise of the resulting product; and
- an outer iterative loop to adjust the specification based on the resulting operational experience, and to rederive, reoptimize, and exercise the adjusted software product.

The transform model thus bypasses the difficulty of having to modify code that has be-come poorly structured through repeated reoptimizations, since the modifications are made to the specification. It also avoids the extra time and expense involved in the inter-mediate design, code, and test activities.

Still, the transform model has various difficulties. Automatic transformation capabilities are only available for small products in a few limited areas: spreadsheets, small fourth-generation language applications, and limited computer science domains. The transform model also shares some of the difficulties of the evolutionary development model, such as the assumption that users' operational systems will always be flexible enough to support unplanned evolution paths. Additionally, it would face a formidable knowledge-base-maintenance problem in dealing with the rapidly increasing and evolving supply of reusable software components and commercial software products. (Simply consider the problem of tracking the costs, performance, and features of all

commercial database management systems, and automatically choosing the best one to implement each new or changed specification.)

The spiral model

The spiral model of the software process (see Figure 2) has been evolving for several years, based on experience with various refinements of the waterfall model as applied to large government software projects. As will be discussed, the spiral model can accommodate most previous models as special cases and further provides guidance as to which combination of previous models best fits a given software situation. Development of the TRW Software Productivity System (TRW-SPS), described in the next section, is its most complete application to date.

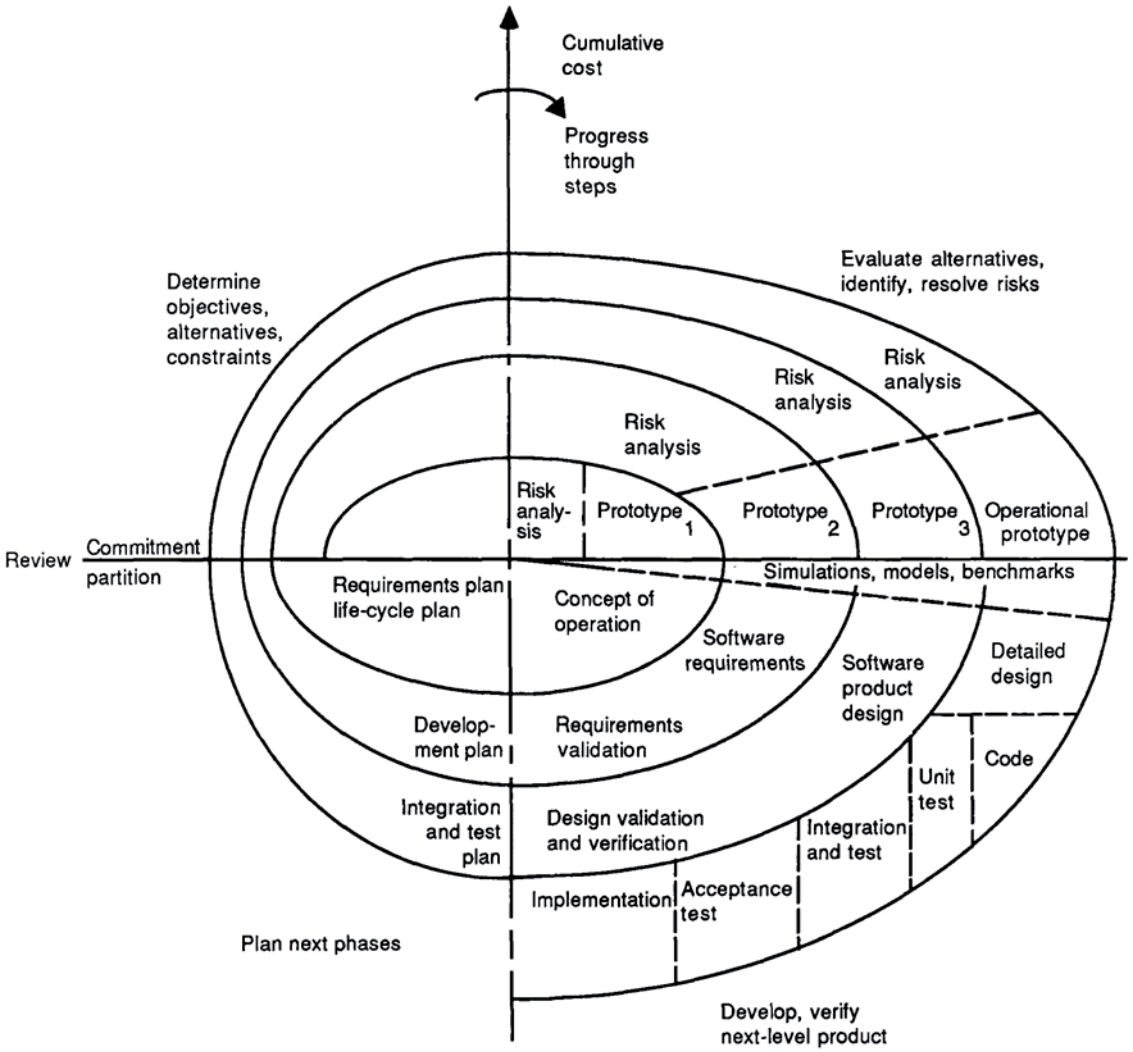


Figure 2. Spiral model of the software process.

The radial dimension in Figure 2 represents the cumulative cost incurred in accomplishing the steps to date; the angular dimension represents the progress made in

completing each cycle of the spiral. (The model reflects the underlying concept that each cycle involves a progression that addresses the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept of operation document down to the coding of each individual program.) Note that some artistic license has been taken with the increasing cumulative cost dimension to enhance legibility of the steps in Figure 2.

A typical cycle of the spiral. Each cycle of the spiral begins with the identification of

- the objectives of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc.);
- the alternative means of implementing this portion of the product (design A , design B, reuse, buy, etc.); and
- the constraints imposed on the application of the alternatives (cost, schedule, inter-face, etc.).

The next step is to evaluate the alternatives relative to the objectives and constraints. Frequently, this process will identify areas of uncertainty that are significant sources of project risk. If so, the next step should involve the formulation of a cost-effective strategy for resolving the sources of risk. This may involve prototyping, simulation, benchmarking, reference checking, administering user questionnaires, analytic modeling, or combinations of these and other risk resolution techniques.

Once the risks are evaluated, the next step is determined by the relative remaining risks. If performance or user-interface risks strongly dominate program development or internal interface-control risks, the next step may be an evolutionary development one: a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping, and the development of a more detailed prototype to continue to resolve the ma-jor risk issues.

If this prototype is operationally useful and robust enough to serve as a low-risk base for future product evolution, the subsequent risk-driven steps would be the evolving series of evolutionary prototypes going toward the right in Figure 2. In this case, the option of writing specifications would be addressed but not exercised. Thus, risk considerations can lead to a project implementing only a subset of all the potential steps in the model.

On the other hand, if previous prototyping efforts have already resolved all of the performance or user-interface risks, and program development or interface-control risks dominate, the next step follows the basic waterfall approach (concept of operation, soft-ware requirements, preliminary design, etc. in Figure 2), modified as appropriate to incorporate incremental development. Each level of software specification in the figure is then followed by a validation step and the preparation of plans for the succeeding cycle.

In this case, the options to prototype, simulate, model, and so on are addressed but not exercised, leading to the use of a different subset of steps.

This risk-driven subsetting of the spiral model steps allows the model to accommodate any appropriate mixture of a specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, on other approach to software development. In such cases, the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks and the relative effectiveness of the various techniques in resolving the risks. In a similar way, risk-management considerations can determine the amount of time and effort that should be devoted to such other project activities as plan-fling, configuration management, quality assurance, formal verification, and testing. In particular, risk-driven specifications (as discussed in the next section) can have varying degrees of completeness, formality, and granularity, depending on the relative risks of doing too little or too much specification.

An important feature of the spinal model, as with most other models, is that each cycle is completed by a review involving the primary people or organizations concerned with the product. This review covers all products developed during the previous cycle, including the plans for the next cycle and the resources required to carry them out. The review's major objective is to ensure that all concerned parties are mutually committed to the approach for the next phase.

The plans for succeeding phases may also include a partition of the product into increments for successive development or components to be developed by individual organizations or persons. For the latter case, visualize a series of parallel spinal cycles, one for each component, adding a third dimension to the concept presented in Figure 2. For example, separate spirals can be evolving for separate software components or increments. Thus, the review-and-commitment step may range from an individual walk-through of the design of a single programmer's component to a major requirements review involving developer, customer, user, and maintenance organizations.

Initiating and terminating the spiral. Four fundamental questions arise in considering this presentation of the spiral model:

(1) How does the spiral ever get started?

(2) How do you get off the spiral when it is appropriate to terminate a project early?

(3) Why does the spiral end so abruptly?

(4) What happens to software enhancement (or maintenance)?

The answers to these questions involve an observation that the spinal model applies equally well to development or enhancement efforts. In either case, the spiral gets started by a hypothesis that a particular operational mission (or set of missions) could be improved by a software effort. The spiral process then involves a test of this hypothesis: at any time, if the hypothesis fails the test (for example, if delays cause a software

product to miss its market window, or if a superior commercial product becomes available), the spirat is terminated. Otherwise, it terminates with the installation of new on modified soft-ware, and the hypothesis is tested by observing the effect on the operational mission. Usually, experience with the operational mission leads to further hypotheses about software improvements, and a new maintenance spiral is initiated to test the hypothesis. Initiation, termination, and iteration of the tasks and products of previous cycles are thus implicitly defined in the spiral model (although they're not included in Figure 2 to simplify its presentation).

Using the spiral model

The various rounds and activities involved in the spinal model are best understood through use of an example. The spiral model was used in the definition and development of the TRW Software Productivity System (TRW-SPS), an integrated software engineering environment.[6] The initial mission opportunity coincided with a corporate initiative to improve productivity in all appropriate corporate operations and an initial hypothesis that software engineering was an attractive area to investigate. This led to a small, extra "Round 0" circuit of the spiral to determine the feasibility of increasing software productivity at a reasonable corporate cost. (Very large or complex software projects will frequently precede the "concept of operation" round of the spiral with one on more smaller rounds to establish feasibility and to reduce the range of alternative solutions quickly and inexpensively.)

Tables 1, 2, and 3 summarize the application of the spiral model to the first three rounds of defining the SPS. The major features of each round are subsequently discussed and are followed by some examples from later rounds, such as preliminary and detailed design.

Table 1. Spiral model usage: TRW Software Productivity System, Round 0

Objectives	Significantly increase software productivity
Constraints	At reasonable cost Within context of TRW culture • Government contracts, high tech, people oriented, security
Alternatives	Management: Project organization, policies, planning, control Personnel: Staffing, incentives, training Technology: Tools, workstations, methods, reuse Facilities: Offices, communications
Risks	May be no high-leverage improvements Improvements may violate constraints
Risk resolution	Internal surveys Analyze cost model Analyze exceptional projects Literature search
Risk resolution results	Some alternatives infeasible • Single time-sharing system: Security Mix of alternatives can produce significant gains

	• Factor of two in five years Need further study to determine best mix
Plan for next phase	Six-person task force for six months More extensive surveys and analysis • Internal, external, economic Develop concept of operation, economic rationale
Commitment	Fund next phase

Round 0: Feasibility study. This study involved five part-time participants over a two-to-three-month period. As indicated in Table 1 , the objectives and constraints were expressed at a very high level and in qualitative terms like "significantly increase," "at reasonable cost," etc.

Some of the alternatives considered, primarily those in the "technology" area, could lead to development of a software product, but the possible attractiveness of a number of non-software alternatives in the management, personnel, and facilities areas could have led to a conclusion not to embank on a software development activity.

The primary risk areas involved possible situations in which the company would invest a good deal only to find that

- Resulting productivity gains were not significant
- Potentially high-leverage improvements were not compatible with some aspects of the "TRW culture"

The risk-resolution activities undertaken in Round 0 were primarily surveys and analyses, including structured interviews of software developers and managers; an initial analysis of productivity leverage factors identified by the constructive cost model (COCOMO);[7] and an analysis of previous projects at TRW exhibiting high levels of productivity.

The risk analysis results indicated that significant productivity gains could be achieved at a reasonable cost by pursuing an integrated set of initiatives in the four major areas. However, some candidate solutions, such as a software support environment based on a single, corporate, maxicomputer-based time-sharing system, were found to be in conflict with TRW constraints requiring support of different levels of security-classified projects. Thus, even at a very high level of generality of objectives and constraints, Round 0 was able to answer basic feasibility questions and eliminate significant classes of candidate solutions.

The plan for Round 1 involved commitment of 12 man-months compared to the two man-months invested in Round 0 (during these rounds, all participants were part-time). Round 1 here corresponded fairly well to the initial round of the spinal model shown in Figure 2, in that its intent was to produce a concept of operation and a basic life-cycle plan for implementing whatever preferred alternative emerged.

Table 2. Spiral model usage: TRW Software Productivity System, Round 1

Objectives	Double software productivity in five years
Constraints	$10,000 per person investment Within context of TRW culture • Government contracts, high tech, people oriented, security Preference for TRW products
Alternatives	Office: Private/modular/… Communication: LAN/star/concentrators/… Terminals: Private/shared; smart/dumb Tools: SREM/PSL-PSA/…; PDL/SADT/… CPU: IBM/DEC/CDC/…
Risks	May miss high-leverage options TRW LAN price/performance Workstation cost
Risk resolution	Extensive external surveys, visits TRW LAN benchmarking Workstation price projections
Risk resolution results	Operations concept: Private offices, TRW LAN, personal terminals, VAX Begin with primarily dumb terminals; experiment with smart workstations Defer operating system, tools selection
Plan for next phase	Partition effort into software development environment (SDE), facilities, management Develop first-cut, prototype SDE • Design-to-cost: 15-person team for one year Plan for external usage
Commitment	Develop prototype SDE Commit an upcoming project to use SDE Commit the SDE to support the project Form representative steering group

Round 1: Concept of operations. Table 2 summarizes Round 1 of the spinal along the lines given in Table 1 for Round 0. The features of Round 1 compare to those of Round 0 as follows:

- The level of investment was greater (12 versus 2 man-months).
- The objectives and constraints were more specific ("double software productivity in five years at a cost of $10,000 a person" versus "significantly increase productivity at a reasonable cost").
- Additional constraints surfaced, such as the preference for TRW products [particularly, a TRW-developed local area network (LAN) system].

- The alternatives were more detailed ("SREM, PSL/PSA on SADT, as requirements tools, etc." versus "tools"; "private/shared" terminals, "smart/dumb" terminals versus "workstations").

- The risk areas identified were more specific ("TRW LAN price-performance within a $10,000-per-person investment constraint" versus "improvements may violate reasonable-cost constraint").

- The risk-resolution activities were more extensive (including the benchmarking and analysis of a prototype TRW LAN being developed for another project).

- The result was a fairly specific operational concept document, involving private of-flees tailored to software work patterns and personal terminals connected to VAX superminis via the TRW LAN. Some choices were specifically deferred to the next round, such as the choice of operating system and specific tools.

- The life-cycle plan and the plan for the next phase involved a partitioning into separate activities to address management improvements, facilities development, and development of the first increment of a software development environment.

- The commitment step involved more than just an agreement with the plan. It committed to apply the environment to an upcoming 100-person testbed software project and to develop an environment focusing on the testbed project's needs. It also specified forming a representative steering group to ensure that the separate activities were well-coordinated and that the environment would not be overly optimized around the testbed project.

Although the plan recommended developing a prototype environment, it also recommended that the project employ requirements specifications and design specifications in a risk-driven way. Thus, the development of the environment followed the succeeding rounds of the spiral model.

Table 3. Spiral model usage: TRW Software Productivity System, Round 2.

Objectives	User-friendly system Integrated software, office-automation tools Support all project personnel Support all life-cycle phases
Constraints	Customer-deliverable SDE ⇨ Portability Stable, reliable service
Alternatives	OS: VMS/AT&T Unix/Berkeley Unix/ISC Host-target/fully portable tool set Workstations: Zenith/LSI-11/…
Risks	Mismatch to user-project needs, priorities

	User-unfriendly system • 12-language syndrome; experts-only Unix performance, support Workstation/mainframe compatibility
Risk resolution	User-project surveys, requirements participation Survey of Unix-using organizations Workstation study
Risk resolution results	Top-level requirements specification Host-target with Unix host Unix-based workstations Build user-friendly front end for Unix Initial focus on tools to support early phases
Plan for next phase	Overall development plan • for tools: SREM, RTT, PDL, office automation tools • for front end: Support tools • for LAN: Equipment, facilities
Commitment	Proceed with plans

Round 2: Top-Level Requirements Specification. Table 3 shows the corresponding steps involved during Round 2 defining the software productivity system. Round 2 decisions and their rationale were covered in earlier work[6]; here, we will summarize the considerations dealing with risk management and the use of the spiral model:

- The initial risk-identification activities during Round 2 showed that several system requirements hinged on the decision between a host-target system or a fully portable tool set and the decision between VMS and Unix as the host operating system. These requirements included the functions needed to provide a user friendly front end, the operating system to be used by the workstations, and the functions necessary to support a host-target operation. To keep these requirements in synchronization with the others, a special minispinal was initiated to address and resolve these issues. The resulting review led to a commitment to a host-target operation using Unix on the host system, at a point early enough to work the OS dependent requirements in a timely fashion.

- Addressing the risks of mismatches to the user-project's needs and priorities resulted in substantial participation of the user-project personnel in the requirements definition activity. This led to several significant redirections of the requirements, particularly toward supporting the early phases of the software life cycle into which the user project was embarking, such as an adaptation of the software requirements engineering methodology (SREM) tools for requirements specification and analysis.

It is also interesting to note that the form of Tables 1, 2, and 3 was originally developed for presentation purposes, but subsequently became a standard "spiral model template" used on later projects. These templates are useful not only for organizing project activities, but also as a residual design-rationale record. Design rationale information is of paramount importance in assessing the potential reusability of software components on future projects. Another important point to note is that the use of the template was indeed uniform across the three cycles, showing that the spiral steps can be and were uniformly followed at successively detailed levels of product definition.

Succeeding rounds. It will be useful to illustrate some examples of how the spinal model is used to handle situations arising in the preliminary design and detailed design of components of the SPS: the preliminary design specification for the requirements traceability tool (RTT), and a detailed design rework on go-back on the unit development folder (UDF) tool.

The RTT preliminary design specification. The RTT establishes the traceability between itemized software requirements specifications, design elements, code elements, and test cases. It also supports various associated query, analysis, and report generation capabilities. The preliminary design specification for the RTT (and most of the other SPS tools) looks different from the usual preliminary design specification, which tends to show a uniform level of elaboration of all components of the design. Instead, the level of detail of the RTT specification is risk-driven.

In areas involving a high risk if the design turned out to be wrong, the design was carried down to the detailed design level, usually with the aid of rapid prototyping. These areas included working out the implications of "undo" options and dealing with the effects of control keys used to escape from various program levels.

In areas involving a moderate risk if the design was wrong, the design was carried down to a preliminary-design level. These areas included the basic command options for the tool and the schemata for the requirements traceability database. Here again, the ease of rapid prototyping with Unix shell scripts supported a good deal of user-interface prototyping.

In areas involving a low risk if the design was wrong, very little design elaboration was done. These areas included details of all the help message options and all the report-generation options, once the nature of these options was established in some example instances.

A detailed design go-back. The UDF tool collects into an electronic "folder" all artifacts involved in the development of a single-programmer software unit (typically 500 to 1,000 instructions): unit requirements, design, code, test cases, test results, and documentation. It also includes a management template for tracking the programmer's scheduled and actual completion of each artifact.

An alternative considered during detailed design of the UDF tool was reuse of portions of the RTT to provide pointers to the requirements and preliminary design

specifications of the unit being developed. This turned out to be an extremely attractive alter-native, not only for avoiding duplicate software development but also for bringing to the surface several issues involving many-to-many mappings between requirements, design, and code that had not been considered in designing the UDF tool. These led to a rethinking of the UDF tool requirements and preliminary design, which avoided a great deal of code rework that would have been necessary if the detailed design of the UDF tool had proceeded in a purely deductive, top-down fashion from the original UDF requirements specification. The resulting go-back led to a significantly different, less costly, and more capable UDF tool, incorporating the RTT in its "uses-hierarchy."

Spiral model features. These two examples illustrate several features of the spiral approach.

- It fosters the development of specifications that are not necessarily uniform, exhaustive, or formal, in that they defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until the high-risk elements of the design are stabilized.

- It incorporates prototyping as a risk reduction option at any stage of development. In fact, prototyping and reuse risk analyses were often used in the process of going from detailed design into code.

- It accommodates reworks on go-backs to earlier stages as more attractive alternatives are identified on as new risk issues need resolution.

Overall, risk-driven documents, particularly specifications and plans, are important features of the spinal model. Great amounts of detail are not necessary unless the absence of such detail jeopardizes the project. In some cases, such as with a product whose functionality may be determined by a choice among commercial products, a set of weighted evaluation criteria for the products may be preferable to a detailed pre-statement of functional requirements.

Results. The Software Productivity System developed and supported using the spiral model avoided the identified risks and achieved most of the system's objectives. The SPS has grown to include over 300 tools and over 1,300,000 instructions; 93 percent of the instructions were reused from previous project-developed, TRW-developed, or external-software packages. Over 25 projects have used all on portions of the system. All of the projects fully using the system have increased their productivity at least 50%; indeed, most have doubled their productivity (when compared with cost-estimation model predictions of their productivity using traditional methods).

However, one risk area—that projects with non-Unix target systems would not accept a Unix-based host system—was underestimated. Some projects accepted the host—target approach, but for various reasons (such as customer constraints and zero-cost target machines) a good many did not. As a result, the system was less widely used on TRW projects than expected. This and other lessons learned have been incorporated

into the spiral model approach to developing TRW's next-generation software development environment.

Evaluation

Advantages. The primary advantage of the spiral model is that its range of options accommodates the good features of existing software process models, while its risk-driven approach avoids many of their difficulties. In appropriate situations, the spiral model becomes equivalent to one of the existing process models. In other situations, it provides guidance on the best mix of existing approaches to a given project; for example, its application to the TRW-SPS provided a risk-driven mix of specifying, prototyping, and evolutionary development.

The primary conditions under which the spiral model becomes equivalent to other main process models are summarized as follows:

- If a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements, and if it has a high risk in budget and schedule predictability and control, then these risk considerations drive the spiral model into an equivalence to the waterfall model.

- If a software product's requirements are very stable (implying a low risk of expensive design and code breakage due to requirements changes during development), and if the presence of errors in the software product constitutes a high risk to the mission it serves, then these risk considerations drive the spiral model to resemble the two-leg model of precise specification and formal deductive program development.

- If a project has a low risk in such areas as losing budget and schedule predictability and control, encountering large-system integration problems, or coping with information sclerosis, and if it has a high risk in such areas as getting the wrong user interface or user decision support requirements, then these risk considerations drive the spinal model into an equivalence to the evolutionary development model.

- If automated software generation capabilities are available, then the spiral model accommodates them either as options for rapid prototyping on for application of the transform model, depending on the risk considerations involved.

- If the high-risk elements of a project involve a mix of the risk items listed above, then the spinal approach will reflect an appropriate mix of the process models above (as exemplified in the TRW-SPS application). In doing so, its risk avoidance features will generally avoid the difficulties of the other models.

The spiral model has a number of additional advantages, summarized as follows:

It focuses early attention on options involving the reuse of existing software. The steps involving the identification and evaluation of alternatives encourage these options.

It accommodates preparation for life-cycle evolution, growth, and changes of the software product. The major sources of product change are included in the prod-uct's objectives, and information-hiding approaches are attractive architectural design alternatives in that they reduce the risk of not being able to accommodate the product-charge objectives.

It provides a mechanism for incorporating software quality objectives into software product development. This mechanism derives from the emphasis on identifying all types of objectives and constraints during each round of the spiral. For example, Table 3 shows user-friendliness, portability, and reliability as specific objectives and constraints to be addressed by the SPS. In Table 1, security constraints were identified as a key risk item for the SPS.

It focuses on eliminating errors and unattractive alternatives early. The risk analysis, validation, and commitment steps cover these considerations.

For each of the sources of project activity and resource expenditure, it answers the key question, "How much is enough?" Stated another way, "How much of requirements analysis, planning, configuration management, quality assurance, testing, formal verification, and so on should a project do?" Using the risk-driven approach, one can see that the answer is not the same for all projects and that the appropriate level of effort is determined by the level of risk incurred by not doing enough.

It does not involve separate approaches for software development and software enhancement (or maintenance). This aspect helps avoid the "second-class citizen" status frequently associated with software maintenance. It also helps avoid many of the problems that currently ensue when high-risk enhancement efforts are approached in the same way as routine maintenance efforts.

It provides a viable framework for integrated hardware-software system development. The focus on risk management and on eliminating unattractive alternatives early and inexpensively is equally applicable to hardware and software.

Difficulties. The full spiral model can be successfully applied in many situations, but some difficulties must be addressed before it can be called a mature, universally applicable model. The three primary challenges involve matching to contract software, relying on risk-assessment expertise, and the need for further elaboration of spiral model steps.

Matching to contract software. The spiral model currently works well on internal software developments like the TRW-SPS, but it needs further work to match it to the world of contract software acquisition.

Internal software developments have a great deal of flexibility and freedom to accommodate stage-by-stage commitments, to defer commitments to specific options, to establish minispinals to resolve critical-path items, to adjust levels of effort, or to accommodate such practices as prototyping, evolutionary development, or design-to-cost. The world of contract software acquisition has a harder time achieving these degrees of flexibility and freedom without losing accountability and control, and a harder time defining contracts whose deliverables are not well specified in advance.

Recently, a good deal of progress has been made in establishing more flexible contract mechanisms, such as the use of competitive front-end contracts for concept definition or prototype fly-offs, the use of level-of-effort and award-fee contracts for evolutionary development, and the use of design-to-cost contracts. Although these have been generally successful, the procedures for using them still need to be worked out to the point that acquisition managers feel fully comfortable using them.

Relying on risk-assessment expertise. The spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk.

A good example of this is the spiral model's risk-driven specification, which carries high-risk elements down to a great deal of detail and leaves low-risk elements to be elaborated in later stages; by this time, there is less risk of breakage.

However, a team of inexperienced or low-balling developers may also produce a specification with a different pattern of variation in levels of detail: a great elaboration of detail for the well-understood, low-risk elements, and little elaboration of the poorly under stood, high-risk elements. Unless there is an insightful review of such a specification by experienced development or acquisition personnel, this type of project will give an illusion of progress during a period in which it is actually heading for disaster.

Another concern is that a risk-driven specification will also be people-dependent. For example, a design produced by an expert may be implemented by non-experts. In this case, the expert, who does not need a great deal of detailed documentation, must produce enough additional documentation to keep the non-experts from going astray. Reviewers of the specification must also be sensitive to these concerns.

With a conventional, document-driven approach, the requirement to carry all aspects of the specification to a uniform level of detail eliminates some potential problems and permits adequate review of some aspects by inexperienced reviewers. But it also creates a large drain on the time of the scarce experts, who must dig for the critical issues within a large mass of non-critical detail. Furthermore, if the high-risk elements have been glossed oven by impressive-sounding references to poorly understood capabilities (such as a new synchronization concept or a commercial DBMS), there is an even greater risk that the conventional approach will give the illusion of progress in situations that are actually heading for disaster.

Need for further elaboration of spiral model steps. In general, the spiral model process steps need further elaboration to ensure that all software development participants are operating in a consistent context.

Some examples of this are the need for more detailed definitions of the nature of spiral model specifications and milestones, the nature and objectives of spiral model reviews, techniques for estimating and synchronizing schedules, and the nature of spiral model status indicators and cost-versus-progress tracking procedures. Another need is for guidelines and checklists to identify the most likely sources of project risk and the most effective risk-resolution techniques for each source of risk.

Highly experienced people can successfully use the spiral approach without these elaborations. However, for large-scale use in situations in which people bring widely differing experience bases to the project, added levels of elaboration—such as have been accumulated oven the years for document-driven approaches—are important in ensuring consistent interpretation and use of the spinal approach across the project.

Efforts to apply and refine the spiral model have focused on creating a discipline of software risk management, including techniques for risk identification, risk analysis, risk prioritization, risk management planning, and risk-element tracking. The prioritized top-ten list of software risk items given in Table 4 is one result of this activity. Another exam-pie is the risk management plan discussed in the next section.

Table 4. A prioritized top-ten list of software risk items

Risk Item	Risk management techniques
1. Personnel shortfalls	Staffing with top talent, job matching; teambuilding; morale building; cross-training; pre-scheduling key people
2. Unrealistic schedules and budgets	Detailed, multisource cost and schedule estimation; design to cost; incremental development; software reuse; requirements scrubbing
3. Developing the wrong software functions	Organization analysis; mission analysis; ops-concept formulation; user surveys; prototyping; early users' manuals
4. Developing the wrong user interface	Task analysis; prototyping; scenarios; user characterization (functionality, style, workload)
5. Gold plating	Requirements scrubbing; prototyping; cost-benefit analysis; design to cost
6. Continuing stream of requirement changes	High change threshold; information hiding; incremental development (defer changes to later increments)
7. Shortfalls in externally furnished components	Benchmarking; inspections; reference checking; compatibility analysis

8.	Shortfalls in externally performed tasks	Reference checking; pre-award audits; award-fee contracts; competitive design or prototyping; teambuilding
9.	Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping; instrumentation; tuning
10.	Straining computer-science capabilities	Technical analysis; cost—benefit analysis; prototyping; reference checking

Implications: The Risk Management Plan. Even if an organization is not ready to adopt the entire spiral approach, one characteristic technique that can easily be adapted to any life-cycle model provides many of the benefits of the spiral approach. This is the risk management plan summarized in Table 5. This plan basically ensures that each project makes an early identification of its top risk items (the number 10 is not an absolute requirement), develops a strategy for resolving the risk items, identifies and sets down an agenda to resolve new risk items as they surface, and highlights progress versus plans in monthly reviews.

Table 5. Software risk management plan

1.	Identify the project's top 10 risk items.
2.	Present a plan for resolving each risk item.
3.	Update list of top risk items, plan, and results monthly.
4.	Highlight risk-item status in monthly project reviews. • Compare with previous month's rankings, status.
5.	Initiate appropriate corrective actions.

The risk management plan has been used successfully at TRW and other organizations. Its use has ensured appropriate focus on early prototyping, simulation, benchmarking, key-person staffing measures, and other early risk-resolution techniques that have helped avoid many potential project "show-stoppers." The recent US Department of Defense standard on software management, DoD-Std-2167, requires that developers produce and use risk management plans, as does its counterpart US Air Force regulation, AFR 800-14.

Overall, the Risk Management Plan and the maturing set of techniques for software risk management provide a foundation for tailoring spinal model concepts into the more established software acquisition and development procedures.

We can draw four conclusions from the data presented:

(1) The risk-driven nature of the spiral model is more adaptable to the full range of software project situations than are the primarily document-driven approaches such as the waterfall model or the primarily code-driven approaches such as evolutionary development. It is particularly applicable to very large, complex, ambitious soft-wane systems.

(2) The spiral model has been quite successful in its largest application to date: the development and enhancement of the TRW-SPS. Overall, it achieved a high level of software support environment capability in a very short time and provided the flexibility necessary to accommodate a high dynamic range of technical alternatives and user objectives.

(3) The spiral model is not yet as fully elaborated as the more established models. Therefore, the spinal model can be applied by experienced personnel, but it needs further elaboration in such areas as contracting, specifications, milestones, reviews, scheduling, status monitoring, and risk area identification to be fully usable in all situations.

(4) Partial implementations of the spiral model, such as the risk management plan, are compatible with most current process models and are very helpful in overcoming major sources of project risk.

Acknowledgments

I would like to thank Frank Belz, Lob Penedo, George Spadano, Bob Williams, Bob Balzen, Gillian Frewin, Peter Hamer, Manny Lehman, Lee Ostenweil, Dave Parnas, Bill Riddle, Steve Squires, and Dick Thayen, along with the Computer reviewers of this article, for their stimulating and insightful comments and discussions of earlier versions, and Nancy Donato for producing its several versions.

References

1. F.P. Brooks et al., *Defense Science Board Task Force Report on Military Software*, Office of the Under Secretary of Defense for Acquisition, Washington, DC 20301, Sept. 1987.

2. H.D. Benington, “Production of Large Computer Programs,” *Proc. ONR Symp. Advanced Programming Methods for Digital Computers*, June 1956, pp. 15–27. Also available in *Annals of the History of Computing*, Oct. 1983, pp. 350–361, and *Proc. Ninth Int’l Conf Software Engineering*, Computer Society Press, 1987.

3. W.W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques,” *Proc. Wescon*, Aug. 1970. Also available in *Proc. ICSE 9*, Computer Society Press, 1987.

4. D.D. McCracken and M.A. Jackson, “Life-Cycle Concept Considered Harmful,” *ACM Software Engineering Notes*, Apr. 1982, pp. 29–32.

5. R. Balzer, T.E. Cheatham, and C. Green, “Software Technology in the I990s: Using a New Paradigm,” *Computer*, Nov. 1983, pp. 39–45.

6. B.W. Boehm et al., “A Software Development Environment for Improving Productivity,” *Computer*, June 1984, pp. 30–44.

7. B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981, Chap. 33.

Further reading

The software process model field has an interesting history, and a great deal of stimulating work has been produced recently in this specialized area. Besides the references to this article, here are some additional good sources of insight:

Overall process model issues and results

Agresti’s tutorial volume provides a good overview and set of key articles. The three recent *Software Process Workshop Proceedings* provide access to much of the recent work in the area.

Agresti, W.W., *New Paradigms for Software Development*, IEEE Catalog No. EH0245-1, 1986.

Dowson, M., ed., *Proc. Third Int’l Software Process Workshop*, IEEE Catalog No. TH0184-2, Nov. 1986.

Potts, C., ed., *Proc. Software Process Workshop*, IEEE Catalog No. 84CH2044-6, Feb. 1984.

Wileden, J.C., and M. Dowson, eds., Proc. Int’l Workshop Software Process and Software Environments, *ACM Software Engineering Notes*, Aug. 1986.

Alternative process models

More detailed information on waterfall-type approaches is given in:

Evans, M.W., P. Piazza, and J.P. Dolkas, *Principles of Productive Software Management*, John Wiley & Sons, 1983.

Hice, G.F., W.J. Turner, and L.F. Cashwell, *System Development Methodology*, North Holland, 1974 (2nd ed., 1981).

More detailed information on evolutionary development is provided in:

Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley, 1988 (currently in publication).

Some additional process model approaches with useful features and insights may be found in:

Lehman, M.M., and L.A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1985.

Osterweil, L., "Software Processes are Software, Too," *Proc. ICSE 9*, IEEE Catalog No. 87CH2432-3, Mar. 1987, pp. 2–13.

Radice, R.A., et al., "A Programming Process Architecture," *IBM Systems J.*, Vol. 24, No. 2, 1985, pp. 79–90.

Spiral and spiral-type models

Some further treatments of spiral model issues and practices are:

Belz, F.C., "Applying the Spiral Model: Observations on Developing System Software in Ada," *Proc. 1986 Annual Conf. on Ada Technology*, Atlanta, 1986, pp. 57–66.

Boehm, B.W., and F.C. Belz, "Applying Process Programming to the Spiral Model," *Proc. Fourth Software Process Workshop*, IEEE, May 1988.

Iivari, J., "A Hierarchical Spiral Model for the Software Process," *ACM Software Engineering Notes*, Jan. 1987, pp. 33–37.

Some similar cyclic spinal-type process models from other fields are described in:

Carlsson, B., P. Keane, and J.B. Martin, "R&D Organizations as Learning Systems," *Sloan Management Review*, Spring 1976, pp. 1–15.

Fisher, R., and W. Ury, *Getting to Yes*, Houghton Mifflin, 1981; Penguin Books, 1983, pp. 68–71.

Kolb, D.A., "On Management and the Learning Process," MIT Sloan School Working Article 652-73, Cambridge, Mass., 1973.

Software risk management

The discipline of software risk management provides a bridge between spiral model concepts and currently established software acquisition and development procedures.

Boehm, B.W., “Software Risk Management Tutorial,” Computer Society, Apr. 1988.

Risk Assessment Techniques, Defense Systems Management College, Ft. Belvoir, Va. 22060, July 1983.

Barry W. Boehm is the chief scientist of the TRW Defense Systems Group. Since 1973, he has been responsible for developing TRW’s software technology base. His current primary responsibilities are in the areas of software environments, process models, management methods, Ada, and cost estimation. He is also an adjunct professor at UCLA.

Boehm received his BA degree in mathematics from Harvard in 1957 and his MA and PhD from UCLA in 1961 and 1964, respectively.

Artikel 2 bij leereenheid 1

Bridging Agile and Traditional Development Methods: A Project Management Perspective
Paul E. McMahon

Systems and Software Technology Conference, April 2004

Bridging Agile and Traditional Development Methods: A Project Management Perspective

Paul E. McMahon
PEM Systems

Monday, 19 April 2004
Track 4: 3:35 - 4:20
Ballroom D

Today, companies are reporting success in meeting rapidly changing customer needs through agile development methods. Many of these same companies are finding they must collaborate with organizations employing more traditional development processes, especially on large Department of Defense projects. While it has been argued that agile methods are compatible with traditional disciplined processes, actual project experience indicates conflicts can arise. This article identifies specific project management conflicts that companies face based on actual project experience, along with strategies employed to resolve these conflicts and reduce related risks. Rationale, insights, and related published references are provided along with lessons learned and recommendations. If you work for a company that is using or considering using agile development, or your company is collaborating with a company using an agile method, this article will help you understand the risks, issues, and strategies that can help your project and organization succeed.

This article was motivated by a case study where a small company using a well-known agile method – eXtreme Programming (XP) – requested help addressing specific conflicts that arose on the project where they were a subcontractor to a larger organization employing a traditional development method. The purpose of this article is not to compare agile and traditional methods, but to raise awareness of potential project management conflicts that can arise when a company employing an agile method collaborates with a company employing a traditional development methodology. It also identifies practical steps that can be taken to reduce related risks.

It is worth noting that the case study presented is not unique. Published references documenting similar conflicts are provided. Also notable is that the motivation for examining this project extends beyond the case study itself. Today there exist increasing opportunities for small companies to gain new work through software outsourcing from traditional development organizations.

Where Are We Going?

In this article, I first identify key case study facts along with relevant information and common misperceptions related to traditional and agile methods. Next, I identify four conflicts observed along with five recommendations and one lesson learned. The company named SubComp refers to the subcontractor employing an agile methodology. The company named PrimeComp refers to the prime contractor employing a traditional development methodology.

Case Study Key Facts

Shortly before I was asked to help SubComp, PrimeComp's customer had withheld a progress payment based on a perceived risk observed at a recent critical design review (CDR). Written comments provided to PrimeComp indicated that the customer wanted to see working software in order to *assess the proposed design and related risk*. The area of concern was SubComp's responsibility. Upon receiving the customer comments, PrimeComp requested that SubComp provide additional detailed design documentation.

It is important to note that PrimeComp required all correspondence between the customer and SubComp to go through them. It is also important to note that most of the contractually required documentation was not formally due until the end of the project, and, prior to the CDR, little had been communicated to SubComp with respect to documentation content and expectations. The project was planned using a traditional waterfall life cycle with a single CDR.

Early in the project, SubComp had identified multiple technical risks. However, it had decided in the early stages to focus its small agile team on a single technical risk that it had assessed to be of much greater significance than all other risks. At the recent CDR, SubComp had provided a demonstration with working software that addressed this risk to the customer's satisfaction.

The risk the customer was currently raising was one SubComp viewed as lower priority. To address this risk, the XP team was focusing on a second demonstration with working software to show to the customer at a follow-up CDR. In parallel, it was also driving to meet PrimeComp's request for additional detailed design documentation. This follow-up CDR had not originally been planned, and it was causing project tension because of the progress payment holdup.

Agile Development Methods

In the spring of 2001, 17 advocates of agile development methods gathered in Utah and agreed to a set of four values and 12 principles referred to as the Agile Manifesto [1]. The four values are expressed in the following:

> We are uncovering better ways of developing software by doing it and helping others do it … Through this work we have come to value:
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value in the items on the right, we value the items on the left more. [1]

One misperception of agile methods is that they hold little or no value in documentation and plans. Note that the value statements express a relative value of documentation and plans with respect to working software and responding to change.

Traditional Development Methods

The traditional waterfall model is well known, but it is important to understand that it is not an incremental model. Today, Department of Defense policy 5000.1 and 5000.2 [2] strongly encourages using the spiral model for software development. Although the spiral model was first introduced by Barry Boehm in the mid-80s [3], the risk-driven essentials of the model frequently have been misunderstood. To help clarify, Boehm recently identified six spiral essentials [2]:

- Concurrent determination of key artifacts.
- Stakeholder review and commitment.
- Level of effort driven by risk.
- Degree of detail driven by risk.
- Use of anchor-point milestones.
- Emphasis on system and life-cycle artifacts (cost/performance goals, adaptability).

A Perspective on Waterfall, Spiral, and Agile Development

Historically, the waterfall life-cycle model has been closely associated with heavyweight documentation. The spiral model has also historically been misinterpreted as an incremental waterfall model, rather than as a risk-based model as clarified by Boehm. It is important to note the focus on people (individuals, stakeholders), products (working software, key artifacts), and change (responding to change, adaptability) common to both the Agile Manifesto and the spiral essentials.

Methods Compatibility or Conflict?

It would seem from this observation that a company using an agile methodology should be able to successfully collaborate with a company using a traditional development method, especially if the project contained risk. To further support this position, Mark Paulk, co-author of the Software Engineering Institute's Capability Maturity Model® (CMM®), which has been associated with rigorous traditional development methods, has stated, "XP is a disciplined process, and the XP process is clearly well defined. We can thus consider CMM and XP complementary" [4].

Despite this evidence of methods compatibility, a different situation appears to exist in the developmental trenches. This is clearly pointed out by Don Reifer in the following statement made in reference to one of his own studies:

> Instead of trying to make XP work rationally with the firm's existing processes, each side is pointing fingers at the other. No one seems to be trying to apply XP within the SW-CMM context rationally and profitably as the sages have suggested ... XP adherents feel they don't have time for the formality … Process proponents argue … quality will suffer and customer satisfaction will be sacrificed. [5]

One proposal to address this conflict has been put forth by Scott Ambler in the form of a *blocker* who runs *interference* for the team by providing, in Ambler's words, "the documents that the bureaucrats request" [6]. The term blocker essentially means someone whose sole job is to keep non-agile project stakeholders from hindering the agile development team's progress.

"Using an incremental life-cycle model is critical because many of the conflicts observed are rooted in the all up-front thinking that comes with the single-increment waterfall model. Incremental thinking is fundamental to agile methods and crucial to bridging the two methods."

In the following paragraphs we identify four conflicts observed on the PrimeComp case study project.

Conflict 1: Working Software vs. Early Design Documentation

Part of the difficulty faced by SubComp is the conflict between what they perceive the end-customer wants with respect to risk management, and what is being asked for by their immediate-customer, PrimeComp. The end-customer has asked to *see working software*. It appears that the end-customer wants more than a *paper* design to assess the risk, yet PrimeComp is asking SubComp to provide more detailed design documentation.

Conflict 2: Single vs. Multiple-Increment Life Cycle

Assuming SubComp succeeds in addressing the immediate high visibility risk, what if yet another risk pops up at the follow-up CDR? Will there be a follow-up to the follow-up CDR with further delays of progress payments?

Agile teams often tackle tough issues first, as did SubComp. They focus on achieving customer satisfaction through frequent software deliveries based on clear priorities. Agile teams are also usually small and often do not have adequate resources to work multiple risks in parallel. The single iteration through the waterfall model with the planned single CDR milestone was a major project management conflict for the agile team. From a project management perspective, it was a critical conflict since a significant payment was withheld.

Conflict 3: Formal Deliverable Documentation Weight

Hearing that the documentation was not due until the end of the project led me to ask two questions:

- What exactly were the contractual documentation requirements?
- Did SubComp know PrimeComp's documentation expectations?

Companies that employ agile methods tend to provide *lightweight* documentation. This is, at least partially, because they value working software more than documentation. Although large documents are not a requirement of traditional development methods, cultural expectations often tend to the *heavyweight* side.

Waiting to deliver documentation until late in the project creates a potential conflict, especially if expectations have not been set through early communication. Because of the stress being placed on the agile team to complete the demo software and to provide additional detailed documentation, the possibility of using a blocker was considered.

Conflict 4: On-Site Customer Collaboration

Agile methods *embrace* [7] changing requirements, even late in development. During my discussions with SubComp personnel, I discovered that the contractual project requirements were, in the words of one team member, "high level and

® CMM is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

open to many interpretations." When contractor and customer work closely on an agile development project, embracing change is eased through close collaboration. When a prime contractor inserts itself in the middle, effective collaboration can be stifled, creating additional conflict and risk.

During my discussions with SubComp personnel, one member of the agile team said,

> What is difficult from my perspective is that the details they [PrimeComp] are asking for are the lowest risk and lowest value technically … spending time on what they feel they need puts our small team at risk for actually completing the product, which is what ultimately matters, I think.

This statement led me to stop and consider just what ultimately did matter. The agile team was addressing the technical risks in a planned manner, but a key ingredient to effective XP operation was missing. How could the agile team determine and act based on what ultimately mattered to the customer when the customer was not collaborating closely on-site?

Given these four observed conflicts, what strategies make sense and what can be done to bridge agile and traditional development?

Recommendation 1: Plan Collaboratively and Use an Incremental Life Cycle

Some believe those who use agile methods do not follow a plan. This is a misunderstanding. Planning is actually a core principle of agile methods [7]; however, agile teams tend to plan in smaller time increments and more frequently than those who use traditional methods. This distinction has been clarified by the characterization of XP as *planning driven* rather than *plan driven* [8]. Planning takes place with both methods, but with agile methods the focus is on the act of planning, rather than a document.

I recommend for similar projects that the initial planning be done collaboratively, with the prime contractor, subcontractor, and customer working closely. I also recommend that an incremental life-cycle model be employed to aid in aligning the agile subcontractor's work within the overall project schedule. Using an incremental life-cycle model is critical because many of the conflicts observed are rooted in the all up-front thinking that comes with the single-increment waterfall model. Incremental thinking is fundamental to agile methods and crucial to bridging the two methods.

In the case study, it was too late to replan the project with an incremental life cycle, but I did recommend that SubComp step back and re-evaluate their strategy to the upcoming follow-up CDR. The following questions needed to be answered:

- Were there other risks that the customer felt had to be addressed at this point for the project to move forward successfully?
- What else would it take to close the CDR?

We had to find out the customer's CDR completion criteria, but we had to do it within an agile mind-set. That is, quickly and with minimal resources given that the small agile team was already overextended working to complete the demo and the prime contractor's request for additional documentation.

"The frequent feedback from multiple spirals can help your risk mitigation visibility and ultimately your project's success."

Recommendation 2: Use the Spiral Model and Well-Defined Anchor-Point Milestones to Address Risk

If one of the risk-responsible collaborating team members is employing an agile method, a spiral model with well-defined anchor points [2, 9] can go a long way to reduce potential project conflict and risk. This model can help the traditional development prime contractor as well as the agile subcontractor.

Providing working software early to address high-risk areas makes sense, but it is not sufficient to meet all project management needs. If you are the prime contractor, you want to make sure all the risks are being addressed in a timely and prioritized fashion. The frequent feedback from multiple spirals can help your risk mitigation visibility and ultimately your project's success.

From the agile subcontractor's perspective, you want your team to be able to focus and solve the highest priority risks early, but you also want to know what the prime contractor's expectations are along the way. By agreeing to the anchor-point milestones during the early collaborative planning activity, expectations can be made clear on both sides, allowing the project to operate more effectively. One of the reasons an incremental life-cycle model is recommended is because it often leads to earlier communication concerning priorities and risks. When a traditional waterfall life-cycle model is selected, early discussions concerning priorities are often missed.

Recommendation 3: Plan for Multiple Documentation Drops and Use a Bridge Person

One reason collaborative initial planning is recommended is to get discussions going early concerning product deliverables thereby reducing the likelihood of late surprises. In the case study, it was decided not to use a *blocker* to provide the contractual documentation. Discussions led the agile team to the conclusion that the blocker notion brought with it a negative view of documentation. The blocker was seen as someone outside the agile team whose job it was to develop the documentation without *bothering* the team. This was not consistent with SubComp's view of documentation, nor was it consistent with the values expressed in the Agile Manifesto.

Our solution was to use what we called a *bridge* person, rather than a blocker. The bridge person, unlike the blocker, was a member of the agile team who participated in team meetings providing a valuable service to the team by capturing key verbal points and whiteboard sketches thereby providing useful maintainable *lightweight* documentation that would ultimately help both contractors and the customer.

I recommended that multiple drops of documentation be provided to PrimeComp prior to the contractual delivery date to get early feedback and reduce late surprises. Waiting until late in the game to deliver documentation is risky, especially when expectations are uncertain.

Recommendation 4: Find a Way to Make Customer Collaboration Work for Effective Requirements Management

The reason establishing a close collaborative working relationship with the customer was not easy in our case study was because PrimeComp was sensitive to any contact between the end-customer and SubComp. This sensitivity was, at least partially, motivated by the desire to maintain control. It is also possible that uncertainty surrounding how the end-customer would perceive the use of an agile methodology fueled PrimeComp's desire to maintain a separation between

SubComp and the end-customer.

A recommendation I would make today to a prime contractor facing similar situations is to recognize that the key to maintaining real project control is the management of risks associated with the subcontractor's effort. One of the most common risks in similar situations is requirements creep, which often fails to get recognized until late in the project's test phase when the customer starts writing new discrepancies because the product does not meet what they now perceive the requirements to mean.

This situation frequently occurs because the end-customer and product developer (agile subcontractor) fail to collaborate sufficiently in the early stages reaching common agreements on potentially ambiguous requirements statements. In an agile development environment, this risk increases because requirements are often written as *story cards* [7], which have an implicit dependence on face-to-face communication to resolve potential differing requirements interpretations.

One common misperception of agile methods is that the requirements are not controlled since they are allowed to change late in the game. This is a legitimate concern that could also fuel why a prime contractor might want to keep an end-customer away from an agile subcontractor, but it also demonstrates a fundamental misunderstanding of agile methods.

As Craig Larman explains, "Iterative and agile methods embrace change, but not chaos" [10]. The following rule clarifies the distinction: "Once the requests for an iteration have been chosen and it is under way, no external stakeholders may change the work" [10].

If you are a prime contractor, I recommend that you check with your agile subcontractor to ensure they understand this crucial distinction between embracing change and living in chaos. I also recommend that a member of the prime contractor's team be placed on the agile team. Then encourage and support as much customer collaboration as you can between the agile team and the end-customer to help manage your own risk.

If you are an agile subcontractor, you want to demonstrate to the prime contractor that you are effectively managing your allocated requirements. Those employing agile methods often use *story point* [11] charts to depict work remaining and requirements creep. While story points and anchor points are different, they can be used together to help bridge the two methods.

Anchor points can be viewed as spiral model progress checkpoints [2, 9]. One weakness with traditional approaches has been the accuracy of the methods employed to measure progress. Story points provide an objective progress-measurement method based on stories verified through successfully completed tests [11].

At the start of each increment, I recommend that the agile subcontractor provide the prime contractor with a documented list of agreed-to stories to be completed in the upcoming increment. Story point charts can then be used to objectively back up anchor-point progress assessments, leading to improved team communication and trust.

Recommendation 5: Document and Communicate Your Process

I recommended to SubComp that they put together a presentation documenting their agile process from the project management perspective. This presentation would include key terms, roles, and responsibilities. Terms unique to the agile method such as coach, tracker, and metaphor [7] would be mapped to traditional terms such as project manager and architecture.

> *"If you are an agile subcontractor, you want to demonstrate to the prime contractor that you are effectively managing your allocated requirements."*

Recommendations for incremental lifecycle model, contract deliverables, and reviews compatible with both agile and traditional development methods should be included. Key to the presentation is a description of how SubComp's agile method fits within a traditional project management framework using a spiral model focusing on risk management. I then recommended to SubComp that they take every opportunity to communicate the key points in the presentation to PrimeComp, the end-customer, and to other traditional development contractors who might hold potential new business opportunities through software outsourcing.

Lesson Learned

When on-site customer collaboration exists, conflicts associated with vague requirements can often be resolved quickly. However, when the customer is not easily accessible, an agile subcontractor with vague requirements can quickly be placed at great risk.

We have learned that today's multi-contractor collaborative projects often do not lend themselves well to full-time, on-site customer collaboration. However, this does not mean that these projects cannot benefit from agility.

In such cases, I recommend a hybrid agile method with a focus on a more traditional requirements development and management method. Successful hybrid agile methods are not new [8, 12]. Keep in mind that hybrid does not imply all requirements up-front, but it does imply that once an iteration is under way, requirements must remain fixed to avoid chaos.

Conclusion

Today, we know how to manage geographically distributed teams formed from companies with divergent cultures and experiences [13]. Bridging agile and traditional development is the next step for organizations looking to take advantage of increasing new business opportunities through collaboration.

If you are experiencing conflicts similar to what has been described in this article, first examine your lines of communication. Look at your terminology. Are you communicating effectively what it is you do and how you do it? If you heard recently that a customer review did not go well, consider that the cause could be as simple as your agile terminology not connecting to the ear of a listener familiar only with traditional methods.

Allowing late requirements changes can work when your customer is on-site working next to you. But if you do not have an on-site collaborative customer, consider a hybrid approach to avoid major trouble late in the project.

Consider bridging, rather than blocking to meet milestone deliverables. More importantly, consider communicating to your collaborative partners and customers through examples of your products to gain their buy-in early, including the weight of your proposed documentation. Let them know that being agile is not cheating, but is in the best interests of everyone.

While many of the solutions described in this article are similar to those employed on non-agile projects, these solutions should not be taken for granted for two reasons. First, too often on hybrid agile-traditional projects, we find emotion

getting in the way of clear thinking, often leading to a fundamental breakdown of communication. Second, we are learning through agile methods more effective techniques to measure progress and communicate. As Robert Martin pointed out in referring to agile methods:

> They're not a regression to the cave, nor are they anything terribly new: Plain and simple, the agile bottom line is the production of regular, reliable data – and that's a good thing. [11]

Don Reifer said, "No one seems to be trying to apply XP within the SW-CMM context rationally and profitably as the sages have suggested" [5]. In our case, studying the proactive steps taken based on the recommendations led to a successful follow-up CDR and to improved communication and early documentation agreements. Today, SubComp recognizes the value of XP, but they also recognize the value and need for fundamental project management, and they are looking to the CMM Integration℠ framework [14] to help guide related improvements.

Agility is not counter to effective project management, but agile methods do not provide all of the project management needs necessary for success. Wrap your agile development process in a lightweight project management framework, and watch your communication and collaboration improve and your project and company succeed.◆

References

1. Cockburn, Alistair. Agile Software Development. Addison-Wesley, 2002: 215-218.
2. Boehm, Barry. "Spiral Model as a Tool for Evolutionary Acquisition." CrossTalk May, 2001 <www.stsc.hill.af.mil/crosstalk/2001/05/index.html>.
3. Boehm, Barry. A Spiral Model of Software Development and Enhancement. Proc. of An International Workshop on Software Process and Software Environments, Trabuco Canyon, CA., Mar. 1985.
4. Paulk, Mark. "Extreme Programming from a CMM Perspective." IEEE Software Nov./Dec. 2001: 19-26.
5. Reifer, Don. "XP and the CMM." IEEE Software May/June 2003: 14-15.
6. Ambler, Scott. "Running Interference." Software Development July, 2003: 50-51.
7. Beck, Kent. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000.
8. Boehm, Barry, and Richard Turner. Balancing Agility and Discipline: A Guide for the Perplexed Addison-Wesley, 2003: 33-34, 233.
9. Boehm, Barry, and Daniel Port. "Balancing Discipline and Flexibility With the Spiral Model and MBASE." CrossTalk Dec. 2001 <www. stsc.hill.af.mil/crosstalk/2001/12/boehm.html>.
10. Larman, Craig. Agile and Iterative Development: A Manager's Guide. Addison-Wesley 2003: 14.
11. Martin, Robert C. "The Bottom Line." Software Development Dec. 2003: 42-44.
12. McMahon, Paul E. "Integrating Systems and Software Engineering: What Can Large Organizations Learn From Small Start-Ups?" CrossTalk Oct. 2002: 22-25 <www.stsc.hill.af.mil/crosstalk/2002/10/mcmahon. html>.
13. McMahon, Paul. E. Virtual Project Management: Software Solutions for Today and the Future. St. Lucie Press, 2001.
14. CMMI Product Team. Capability Maturity Model® Integration (CMMI®), Version 1.1. Pittsburgh, PA: Software Engineering Institute <www.sei.cmu.edu>.

About the Author

Paul E. McMahon, principal of PEM Systems, provides technical and management services to large and small engineering organizations. He has taught software engineering at Binghamton University; conducted workshops on engineering process and management; and published over 20 articles, including articles on agile development and distributed development in CrossTalk, and a book on collaborative development, "Virtual Project Management: Software Solutions for Today and the Future." McMahon also presented at the 2003 Software Technology Conference on "Growing Effective Technical Managers."

PEM Systems
118 Matthews ST
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org

Artikel 3 bij leereenheid 1

Improving Software Economics
Walker Royce

Whitepaper, IBM Rational Software, 2009

Improving Software Economics

Top 10 Principles of Achieving Agility at Scale

by Walker Royce
Vice President, IBM Software Services, Rational

Contents

From software development to software delivery

The world is becoming more dependent on software delivery efficiency and world economies are becoming more dependent on producing software with improved economic outcomes. What we have learned over decades of advancing software development best practice is that software production involves more of an economics than an engineering discipline. This paper provides a provocative perspective on achieving agile software delivery and the economic foundations of modern best practices.

Improvement in software life-cycle models and software best practices has been a long slog that accelerated in the 1980s as the engineering roots of software management methods continued to fail in delivering acceptable software project performance. IBM's Rational team has partnered with hundreds of software organizations and participated in thousands of software projects over the last 25 years. Our mission has been twofold: first, to bring software best practices to our customers, and second, to participate directly on their diverse projects to learn the patterns of success and failure so that we could differentiate which practices were best, and why. The Rational team didn't invent iterative development, object-oriented design, UML, agile methods, or the best practices captured in the IBM® Rational® Unified Process. The industry evolved these techniques, and we built a business out of synthesizing the industry's experience and packaging lessons learned into modern processes, methods, tools, and training. This paper provides a short history of this transition by looking at the evolution of our management principles. It presents our view of the Top 10 principles in managing an industrial-strength software organization and achieving agility at any scale of business challenge.

Most organizations that depend on software are struggling to transform their life-cycle model from a development focus to a delivery focus. This subtle distinction in wording represents a dramatic change in the principles that are driving the management philosophy and the governance models. Namely, a "software development" orientation focuses on the various activities required in the development process, while a "software delivery" orientation focuses on the results of that process. Organizations that have successfully made this transition—perhaps 30-40% by our estimate—have recognized that engineering discipline is trumped by economics discipline in most software-intensive endeavors.

Table 1 provides a few differentiating indicators of successfully making the transformation from conventional engineering governance to more economically driven governance.

Table 1: Differentiating conventional engineering governance from economically driven governance

Software Development: Engineering Driven	**Software Delivery: Economics Driven**
Distinct development phases	Continuously evolving systems
Distinct handoff from development team to maintenance team	Common process, platform, and team for development and maintenance
Distinct and sequential activities from requirements to design to code to test to operations	Sequence of usable capabilities with ever-increasing value
Phase and role specific processes and tools	Collaborative platform of integrated tools and process enactment
Collocated teams	Geographically distributed, Web-based collaboration
Early precision in complete plans and requirements	Evolving precision as uncertainties are resolved
Governance through measurement of artifact production and activity completion	Governance through measurement of incremental outcomes, and progress/quality trends
Engineering discipline: precisely define requirements/plans completely, then track progress against static plans and scope	Economic discipline: reduce uncertainties, manage variance, measure trends, adapt and steer with continuous negotiation of targets

Success rates in applying engineering governance (a.k.a. waterfall model management) have been very low; most industry studies assess the success rate at 10-20%. Where waterfall model projects do succeed, one usually finds that the project has been managed with two sets of books. The front-office books satisfy the external stakeholders that the engineering governance model is being followed and the back-office books, where more agile techniques are employed with economic governance, satisfy the development team that they can predictably deliver results in the face of the uncertainties. The results of the back-office work gets fed back to meet the deliverables and milestones required for the front-office books. "Managing two sets of books" has been expensive, but it is frequently the only way for developers to deliver a satisfactory product while adhering to the stakeholder

demand for engineering governance. Advanced organizations have transitioned to more efficiently managing only one set of honest plans, measures, and outcomes. Most organizations still manage some mixture of engineering governance and economic governance to succeed.

Let's take a minute to think about engineering vs. economics governance -- i.e., precise up-front planning vs. continual course correction toward a target goal -- in terms even those outside the software industry can relate to. This may be a thought-provoking hypothesis: Software project managers are more likely to succeed if they use techniques similar to those used in movie production, compared to those used conventional engineering projects, like bridge construction.[1,2] Consider this:

- Most software professionals have no laws of physics, or properties of materials, to constrain their problems or solutions. They are bound only by human imagination, economic constraints, and platform performance once they get something executable.
- Quality metrics for software products have few accepted atomic units. With the possible exception of reliability, most aspects of quality are very subjective, such as responsiveness, maintainability and usability. Quality is best measured through the eyes of the audience.
- In a software project, you can seemingly change almost anything at any time: plans, people, funding, requirements, designs, and tests. Requirements—probably the most misused word in our industry—rarely describe anything that is truly required. Nearly everything is negotiable.

These three observations are equally applicable to software project managers and movie producers. These are professionals that

regularly create a unique and complex web of intellectual property bounded only by a vision and human creativity. Both industries experience a very low success rate relative to mature engineering enterprises.

The last point above is worth a deeper look. The best thing about software is that it is soft (i.e., relatively easy to change) but this is also its riskiest attribute. In most systems, the software is where we try to capture and anticipate human behavior, including abstractions and business rules. Most software does not deal with natural phenomena where laws of physics or materials provide a well-understood framework. Hence, most software is constrained only by human imagination; the quality of software is judged more like a beauty contest than by precise mathematics and physical tolerances. If we don't carefully manage software production, we can lull ourselves into malignant cycles of change that result in massive amounts of scrap, rework, and wasted resources.

With the changeability of software being its greatest asset and greatest risk, it is imperative that we measure software change costs and qualities and understand the trends therein. The measure of scrap and rework is an economic concern that has long been understood as a costly variable in traditional engineering, as in the construction industry. While in the software industry we commonly blow up a product late in the lifecycle and incur tremendous scrap and rework to rebuild its architecture, we rarely do this in the construction industry. The costs are so tangibly large, and the economic ramifications are dire. In software, we need to get an equally tangible understanding of the probable economic outcomes.

A lesson that the construction industry learned long ago was to eliminate the risk of reinventing the laws of construction on every

project. Consequently, they enforced standards in building codes, materials, and techniques, particularly for the architectural engineering aspects of structure, power, plumbing, and foundation. This resulted in much more straightforward (i.e., predictable) construction with innovation mostly confined to the design touches sensed by its human users. This led to guided economic governance for the design/style/usability aspects with standardization and engineering governance driving most of the architecture, materials, and labor. When we innovate during the course of planned construction projects with new materials, new technology, or significant architectural deviations, it leads to the same sorts of overruns and rework that we see in software projects. For most products, systems, and services, you want to standardize where you can and not reinvent.

Economic discipline and governance is needed to measure the risk and variance of the uncertain outcomes associated with innovation. Most software organizations undertake a new software project by permitting their most trusted craftsmen to reinvent software capabilities over and over. Each project and each line of business defend the reasons why their application is different, thereby requiring a custom solution without being precise about what is different. Encumbered with more custom developed architectures and components than reused ones, they end up falling back on the waterfall model, which is easy to understand. But this approach is demonstrably too simplistic for uncertain endeavors like software.

The software industry has characterized new and improved software life-cycle models using many different terms, such as: spiral development, incremental development, evolutionary development, iterative development, and agile development. In spirit, these models have many things in common, and, as a class,

they represent a common theme: anti-waterfall development. However, after 20-30 years of improvement and transition, the waterfall model mindset is still the predominant governance process in most industrial-strength software development organizations. By my estimation, more than half of the software projects in our industry still govern with a waterfall process, particularly organizations with mature processes. Perhaps geriatric could be used as an explicit level of process maturity, one that should be recognized in software maturity models to help organizations identify when their process has become too mature and in need of a major overhaul.

The move to agility

We have learned many best practices as we evolved toward modern agile delivery methods. Most of them we discovered years ago as we worked with forward-looking organizations. At IBM, we have been advancing techniques largely from the perspective of industrial strength software engineering, where scale and criticality of applications dominate our governance and management methods. We were one of the pioneers of agile techniques like pair programming[3] and extreme programming,[4] and IBM now has a vibrant technical community with thousands of practitioners engaged in agile practices in our own development efforts and our professional services. Many pioneering teams inside and outside of IBM have advanced these best practices from smaller scale techniques, commonly referred to as "agile methods," and these contributions were developed separately in numerous instances across the diverse spectrum of software domains, scales, and applications.

For years, we have worked to unite the agile consultants (i.e., small scale development camps) with the process maturity

consultants (i.e., industrial strength software development camps). While these camps have been somewhat adversarial and wary of endorsing one another, both sides have valid techniques and a common spirit, but approach common problems with a different jargon and bias. There is no clear right or wrong prescription for the range of solutions needed. Context and scale are important, and every nontrivial project or organization needs a mix of techniques, a family of process variants, common sense, and domain experience to be successful.

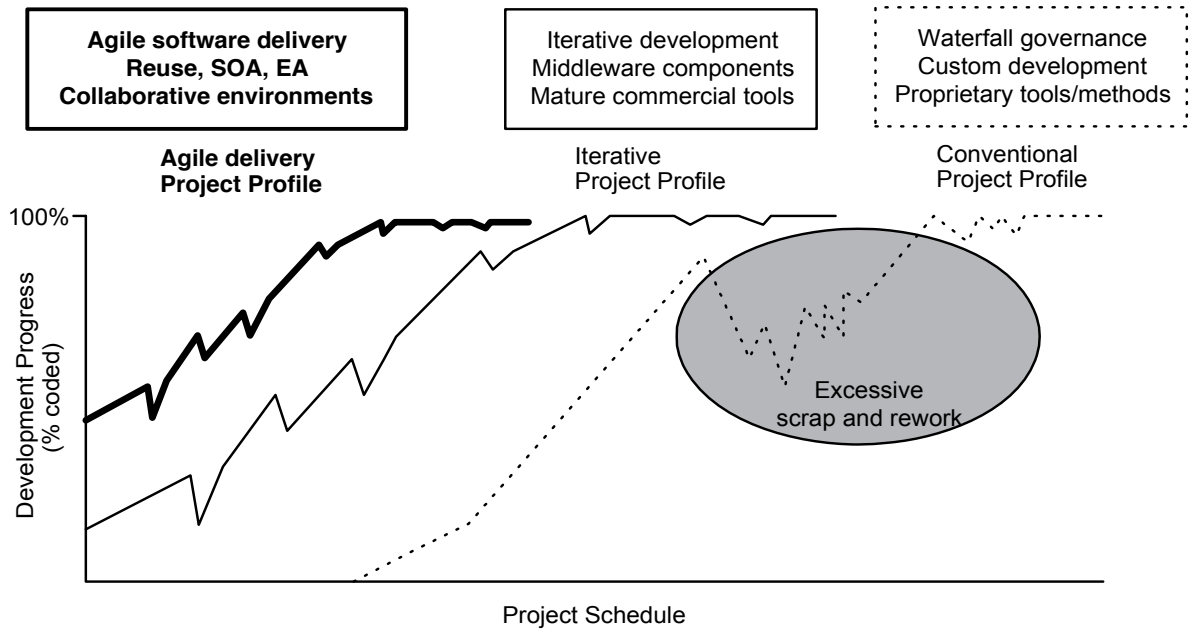
In *Software Project Management,*[5] I introduced my Top 10 Principles of managing a modern software process. I will use that framework to summarize the history of best-practice evolution. The sections that follow describe three discrete eras of software life-cycle models by capturing the evolution of their top 10 principles. I will denote these three stages as 1) conventional waterfall development, 2) transitional iterative development, and 3) modern agile delivery. I will only describe the first two eras briefly since they have been covered elsewhere in greater detail and their description here is only to provide benchmarks for comparison to the top 10 principles of a modern agile delivery approach.

Figure 1 provides a project manager's view of the process transition that the industry has been marching toward for decades. Project profiles representing each of the three eras plot development progress versus time, where progress is defined as percent executable—that is, demonstrable in its target form. Progress in this sense correlates to tangible intermediate outcomes, and is best measured through executable demonstrations. The term "executable" does not imply complete, compliant, nor up to specifications; but it does imply that the software is testable. The figure also describes the primary

measures that were used to govern projects in these eras and introduces the measures that we find to be most important moving forward to achieve agile software delivery success.

Conventional waterfall projects are represented by the dotted line profile in Figure 1. The typical sequence for the conventional waterfall management style when measured this way is:

1. Early success via paper designs and overly precise artifacts,
2. Commitment to executable code late in the life cycle,
3. Integration nightmares due to unforeseen implementation issues and interface ambiguities,
4. Heavy budget and schedule pressure to get the system working,
5. Late shoe-horning of suboptimal fixes, with no time for redesign, and
6. A very fragile, expensive-to-maintain product, delivered late.



Agile Econometrics	Iterative Trends	Waterfall measures
Accurate net present value	Honest earned value	Dishonest earned values
Reuse/custom asset trends	Release content over time	Activity/milestone completion
Release quality over time	Release quality over time	Code/test production
Variance in estimate to complete	Prioritized risk management	Requirements-design-code traceability
Release content/quality over time	Scrap/rework/defect trends	Inspection coverage
Actuals vs dynamic plans	Actuals vs dynamic plans	Actuals vs static plan

Figure 1: Improved project profiles and measures in transitioning to agile delivery processes

Most waterfall projects are mired in inefficient integration and late discovery of substantial design issues, and they expend roughly 40% or more of their total resources in integration and test activities, with much of this effort consumed in excessive scrap and rework during the late stages of the planned project, when project management had imagined shipping or deploying the software. Project management typically reports a linear progression of earned value up to 90% complete before reporting a major increase in the estimated cost of completion as they suffer

through the late scrap and rework. In retrospect, software earned value systems based on conventional activity, document, and milestone completion are not credible since they ignore the uncertainties inherent in the completed work. Here is a situation for which I have never seen a counter-example: A software project that has a consistently increasing progress profile is certain to have a pending cataclysmic regression.

The iterative management approach represented by the middle profile in Figure 1 forces integration into the design phase through a progression of demonstrable releases, thereby exposing the architecturally significant uncertainties to be addressed earlier where they can be resolved efficiently in the context of life-cycle goals. Equally as critical to the process improvements are a greater reliance on more standardized architectures and reuse of operating systems, data management systems, graphical user interfaces, networking protocols, and other middleware. This reuse and architectural conformity contributes significantly to reducing uncertainty through less custom development and precedent patterns of construction. The downstream scrap and rework tarpit is avoidable, along with late patches and malignant software fixes. The result is a more robust and maintainable product delivered more predictably with a higher probability of economic success. Iterative projects can deliver a product with about half the scrap and rework activities as waterfall projects by re-factoring architecturally significant changes far earlier in the lifecycle.

Agile software delivery approaches start projects with an ever increasing amount of the product coming from existing assets, architectures, and services, as represented in the left hand profile. Integrating modern best practices and a supporting platform that enables advanced collaboration allows the team to iterate more effectively and efficiently. Measurable progress and quality are

accelerated and projects can converge on deliverable products that can be released to users and testers earlier. Agile delivery projects that have fully transitioned to a steering leadership style based on effective measurement can optimize scope, design, and plans to reduce this waste of unnecessary scrap and rework further, eliminate uncertainties earlier, and significantly improve the probability of win-win outcomes for all stakeholders. Note that we don't expect scrap and rework rates to be driven to zero, but rather to a level that corresponds to healthy discovery, experimentation, and production levels commensurate with resolving the uncertainty of the product being developed.

Table 2 provides one indicative benchmark of this transition. The resource expenditure trends become more balanced across the primary workflows of a software project as a result of less human-generated stuff, more efficient processes (less scrap and rework), more efficient people (more creative work, less overhead), and more automation.

Table 2: Resource expenditure profiles in transitioning to agile delivery processes

Lifecycle Activity	Conventional	Iterative	Agile
Management	5%	10%	**10-15%**
Scoping	5%	10%	**10-15%**
Design/demonstration	10%	15%	**10-20%**
Implementation/coding	30%	25%	**15-20%**
Test and assessment	40%	25%	**15-25%**
Release and Deployment	5%	5%	**10%**
Environment/tooling	5%	10%	**10%**

Top 10 Principles of Conventional Software Management

Most software engineering references present the waterfall model[6] as the source of the "conventional" software management process, and I use these terms interchangeably. Years ago, I asserted the top 10 principles of the conventional software process to capture its spirit and provide a benchmark for comparison with modern methods. The interpretation of these principles and their order of importance are judgments that I made based on experiences from 100s of project evaluations, project diagnoses performed by the Rational team, and discussions with Winston Royce, one of the pioneers in software management processes. My father is well-known for his work on the waterfall model, but he was always more passionate about iterative and agile techniques well before they became popular.[7]

Top 10 Management Principles of Waterfall Development

1. Freeze requirements before design.
2. Forbid coding prior to detailed design review.
3. Use a higher order programming language.
4. Complete unit testing before integration.
5. Maintain detailed traceability among all artifacts.
6. Thoroughly document each stage of the design.
7. Assess quality with an independent team.
8. Inspect everything.
9. Plan everything early with high fidelity.
10. Control source code baselines rigorously.

Conventional software management techniques typically follow a sequential transition from requirements to design to code to test with extensive paper-based artifacts that attempt to capture

complete intermediate representations at every stage. Requirements are first captured in complete detail in ad hoc text and then design documents are fully elaborated in ad hoc notations. After coding and unit testing individual code units, they are integrated together into a complete system. This integration activity is the first time that significant inconsistencies among components (their interfaces and behavior) can be tangibly exposed, and many of them are extremely difficult to resolve. Integration — getting the software to operate reliably enough to test its usefulness — almost always takes much longer than planned. Budget and schedule pressures drive teams to shoehorn in the quickest fixes. Re-factoring the design or reconsideration of requirements is usually out of the question. Testing of system threads, operational usefulness, and requirements compliance gets performed through a series of releases until the software is judged adequate for the user. More than 80% of the time, the end result is a late, over-budget, fragile, and expensive-to-maintain software system.

Hindsight from thousands of software project post-mortems has revealed a common symptom of governing a software project with an engineering management style: the project's integration and test activities require an excessive expenditure of resources in time and effort. This excessive rework is predominantly a result of postponing the resolution of architecturally significant issues (i.e., resolving the more serious requirements and design uncertainties) until the integration and test phase. We observed that better performing projects would be completed with about 40% of their effort spent in integration and test. Unsuccessful projects spent even more. With less than one in five projects succeeding, better governance methods were imperative.

One of the most common failure patterns in the software industry is to develop a five-digits-of-precision version of a requirement specification (or plan) when you have only a one-digit-of-precision understanding of the problem. A prolonged effort to build precise requirements or a detailed plan only delays a more thorough understanding of the architecturally significant issues — that is, the essential structure of a system and its primary behaviors, interfaces, and design trade-offs. How many frighteningly thick requirements documents or highly precise plans (i.e., inchstones rather than milestones) have you worked on, perfected, and painstakingly reviewed, only to completely overhaul these documents months later?

The single most important lesson learned in managing software projects with the waterfall model was that software projects contain much more uncertainty than can be accommodated with an engineering governance approach. This traditional approach presumes well-understood requirements and straightforward production activities based on mature engineering precedent.

Top 10 Principles of Iterative Software Management

In the 1990s, Rational Software Corporation began evolving a modern process framework to more formally capture the best practices of iterative development. The primary goal was to help the industry transition from a "plan and track" management style (the waterfall model) to a "steering" leadership style that admitted uncertainties in the requirements, design, and plans.

The software management approach we evolved led to producing the architecture first, then usable increments of partial capability, then you worry about completeness. Requirements and design flaws are detected and resolved earlier in the life cycle, avoiding the big-bang integration at the end of a project by integrating in

stages throughout the project life cycle. Modern, iterative development enables better insight into quality because system characteristics that are largely inherent in the architecture (e.g., performance, fault tolerance, adaptability, interoperability, maintainability) are tangible earlier in the process where issues are still correctable without jeopardizing target costs and schedules. These techniques attacked major uncertainties far earlier and more effectively. Here are my top 10 principles of iterative development[8] from the 1990s and early 2000s era:

Top 10 Management Principles of Iterative Development

1. Base the process on an architecture-first approach.
2. Establish an iterative life-cycle process that confronts risk early.
3. Transition design methods to emphasize component-based development.
4. Establish a change management environment.
5. Enhance change freedom through tools that support round-trip engineering.
6. Capture design artifacts in rigorous, model-based notation.
7. Instrument the process for objective quality control and progress assessment.
8. Use a demonstration-based approach to assess intermediate artifacts.
9. Plan intermediate releases in groups of usage scenarios with evolving levels of detail.
10. Establish a configurable process that is economically scalable.

Whereas conventional principles drove software development activities to overexpend in integration activities, these modern principles resulted in less total scrap and rework through relatively

more emphasis in early life-cycle engineering and a more balanced expenditure of resources across the core workflows of a modern process.

The architecture-first approach forces integration into the design phase, where the most significant uncertainties can be exposed and resolved. The early demonstrations do not eliminate the design breakage; they just make it happen when it can be addressed effectively. The downstream scrap and rework is significantly reduced along with late patches and sub-optimal software fixes, resulting in a more robust and maintainable design.

Interim milestones provide tangible results. Designs are now "guilty until proven innocent." The project does not move forward until the objectives of the demonstration have been achieved. This does not preclude the renegotiation of objectives once the milestone results permit further refactoring and understanding of the tradeoffs inherent in the requirements, design, and plans.

Figure 1 illustrates the change in measurement mindset when moving from waterfall model measures of activities to iterative measures of scrap and rework trends in executable releases. The trends in *cost of change*[9] can be observed through measuring the complexity of change This requires a project to quantify the rework (effort required for resolution) and number of instances of rework. In simple terms, adaptability quantifies the ease of changing a software baseline, with a lower value being better. When changes are easy to implement, a project is more likely to increase the number of changes, thereby increasing quality. With the conventional process and custom architectures, change was more expensive to incorporate as we proceeded later into the life cycle. For waterfall projects that measured such trends, they tended to see the cost of change increase as they transitioned

from testing individual units of software to testing the larger, integrated system. This is intuitively easy to understand, since unit changes (typically implementation issues or coding errors) were relatively easy to debug and resolve and integration changes (design issues, interface errors or performance issues) were relatively complicated to resolve.

A discriminating result of a successful transition to a modern iterative process with an architecture first approach is that the more expensive changes are discovered earlier when they can be efficiently resolved and get simpler and more predictable as we progress later into the life cycle. This is the result of attacking the uncertainties in architecturally significant requirements tradeoffs and design decisions earlier. The big change in an iterative approach is that integration activities mostly precede unit test activities, thereby resolving the riskier architectural and design challenges prior to investing in unit test coverage and complete implementations.

This is the single most important measure of software project health. If you have a good architecture and an efficient process, the long-accepted adage: "*The later you are in the life cycle, the more expensive things are to fix*" does NOT apply. [10]

Successful steering in iterative development is based on improved measurement and metrics extracted directly from the evolving sequence of executable releases. These measures, and the focus on building the architecture first, allow the team to explicitly assess trends in progress and quality and systematically address the primary sources of uncertainty. The absolute measures are useful, but the relative measures (or trends) of how progress and quality change over time are the real discriminators in improved steering, governance, and predictability.

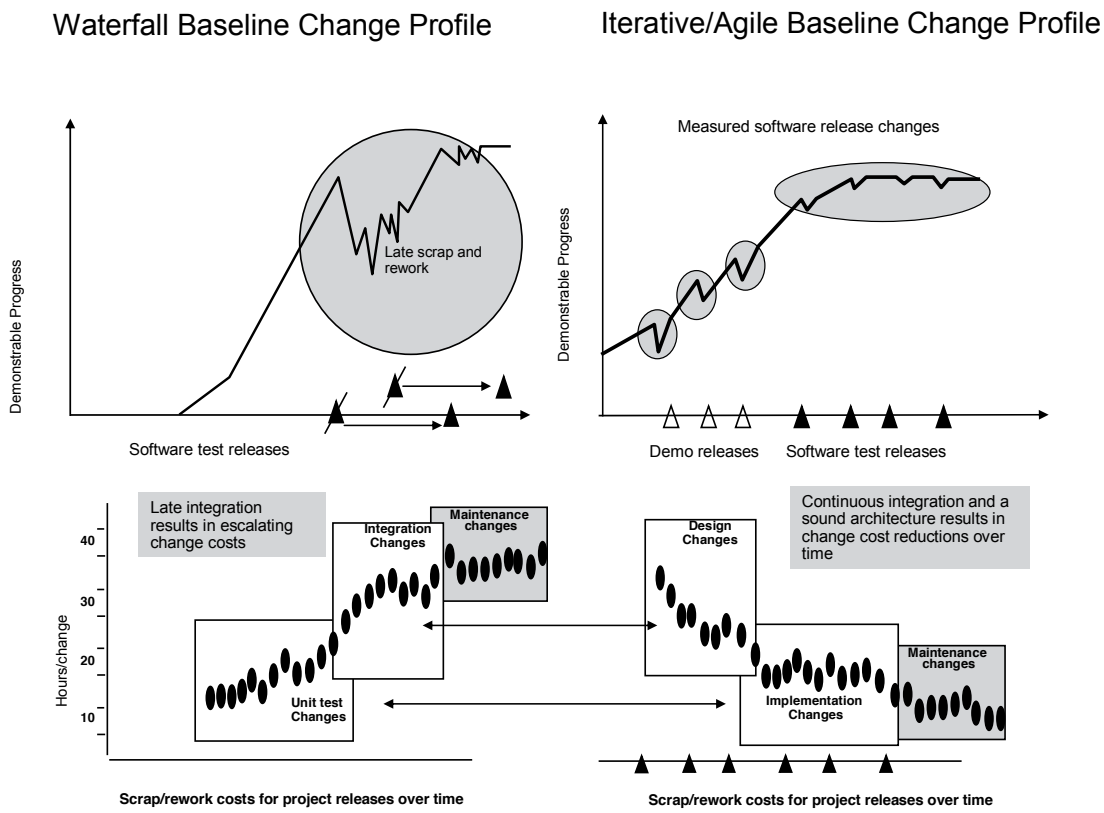


Figure 1: The discriminating improvement measure: change cost trends

Balancing innovation with standardization is critical to governing the cost of iterating, as well as governing the extent to which you can reuse assets versus developing more custom components. Standardization through reuse can take on many forms including:

- **Product assets**: architectures, patterns, services, applications, models, commercial components, legacy systems, legacy components…
- **Process assets**: methods, processes, practices, measures, plans, estimation models, artifact templates …

- **People assets**: existing staff skills, partners, roles, ramp-up plans, training …
- **Platform assets**: schemas, commercial tools, custom tools, data sets, tool integrations, scripts, portals, test suites, metrics experience databases …

While this paper is primarily concerned with the practice of reducing uncertainty, there is an equally important practice of reusing assets based on standardization. The value of standardizing and reusing existing architectural patterns, components, data, and services lies in the reduction in uncertainty that comes from using elements whose function, behavior, constraints, performance, and quality are all known. The cost of standardizing and reuse is that it can constrain innovation. It is therefore important to balance innovation and standardization, which requires emphasis on economic governance to reduce uncertainty; but that practice is outside the scope of this paper.

Reducing uncertainty: The basis of best practice

The top 10 principles of iterative development resulted in many best practices, which are documented in the Rational Unified Process.[11] The Rational Unified Process includes practices for requirements management, project management, change management, architecture, design and construction, quality management, documentation, metrics, defect tracking, and many more. These best practices are also context dependent. For example, a specific best practice used by a small research and development team at an ISV is not necessarily a best practice for an embedded application built to military standards. After several years of deploying these principles and capturing a framework of

best practices, we began to ask a simple question: "Why are these best? And what makes them better?"
IBM research and the IBM Rational organization have been analyzing these questions for over a decade, and we have concluded that *reducing uncertainty* is THE recurring theme that ties together techniques that we call best practices. Here is a simple story that Murray Cantor composed to illustrate this conclusion.

Suppose you are the assigned project manager for a software product that your organization needs to be delivered in 12 months to satisfy a critical business need. You analyze the project scope and develop an initial plan and mobilize the project resources estimated by your team. They come back after running their empirical cost/schedule estimation models and tell you that the project should take 11 months. Excellent! What do you do with that information? As a savvy and scarred software manager, you know that the model's output is just a point estimate and simply the expected value of a more complex random variable, and you would like to understand the variability among all the input parameters and see the full distribution of possible outcomes. You want to go into this project with a 95% chance of delivering within 12 months. Your team comes back and shows you the complete distribution illustrated as the "baseline estimate" at the top of Figure 2. I'll describe the three options shown in a moment.

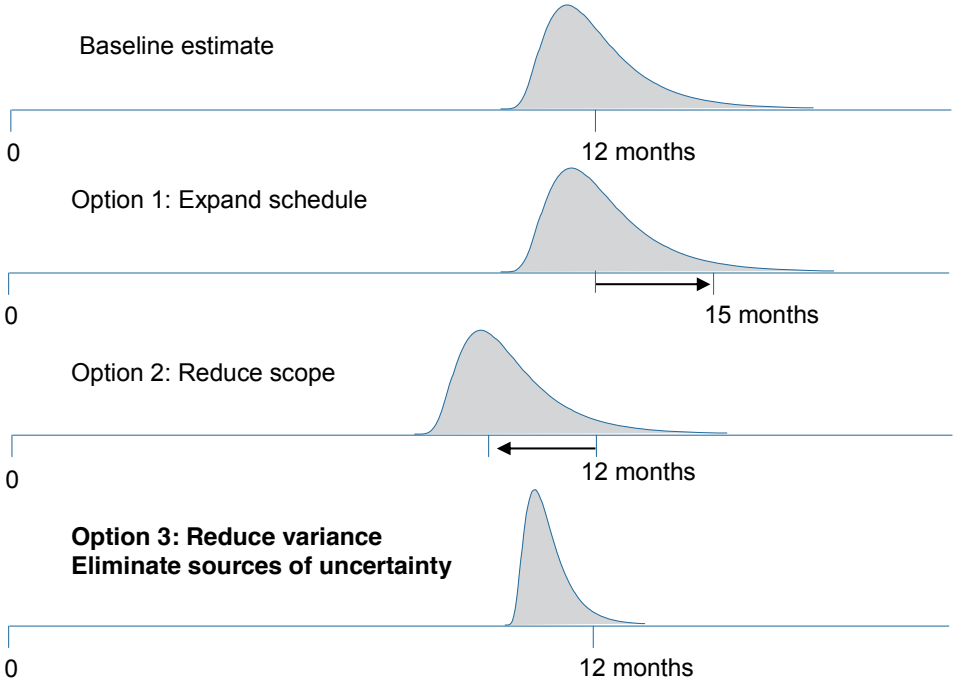


Figure 2: A baseline estimate and alternatives in dealing with project management constraints.

Examining the baseline estimate, you realize that about half of the outcomes will take longer than 12 months and you have only about a 50% chance of delivering on time. The reason for this dispersion is the significant uncertainty in the various input parameters reflecting your team's lack of knowledge about the scope, the design, the plan, and the team capability. Consequently, the variance of the distribution is rather wide.[12] Now, as a project manager there are essentially three paths that you can take; these are also depicted in Figure 2:

1. Option 1: Ask the business to move out the target delivery date to 15 months to ensure that 95% of the outcomes complete in less time than that.

2. Option 2: Ask the business to re-scope the work, eliminating some of the required features or backing off on quality so that the median schedule estimate moves up by a couple of months. This ensures that 95% of the outcomes complete in 12 months.

3. Option 3: This is the usual place we all end up and the project managers that succeed work with their team to shrink the variance of the distribution. You must address and reduce the uncertainties in the scope, the design, the plans, the team, the platform, and the process. The effect of eliminating uncertainty is less dispersion in the distribution and consequently a higher probability of delivering within the target date.

   The first two options are usually deemed unacceptable, leaving the third option as the only alternative—and the foundation of most of the iterative and agile delivery best practices that have evolved in the software industry. If you examine the best practices for requirements management, use case modeling, architectural modeling, automated code production, change management, test management, project management, architectural patterns, reuse, and team collaboration, you will find methods and techniques to reduce uncertainty earlier in the life cycle. If we retrospectively examine my top 10 principles of iterative development, one can easily conclude that many of them (specifically 1, 2, 3, 6, 8, and 9) make a significant contribution to addressing uncertainties earlier. The others (4, 5, 7 and 10) are more concerned with establishing feedback control environments for measurement and reporting. It was not obvious to me that the purpose of these principles was also to reduce uncertainty until I read Douglass Hubbard's book *How to Measure Anything,*[13] where I rediscovered the following definition:

> Measurement: A set of observations that reduce uncertainty where the result is expressed as a quantity.

Voila! The scientific community does not look at measurement as completely eliminating uncertainty. Any significant reduction in uncertainty is enough to make a measurement valuable. With that context, I concluded that the primary discriminator of software delivery best practices was that they effectively reduce uncertainty and thereby increase the probability of success—even if success is defined as cancelling a project earlier so that wasted cost was minimized. What remains to be assessed are how much better these practices work in various domains and how do we best instrument them. IBM research continues to invest in these important questions.

Achieving "Agility at Scale": Top 10 principles of Agile software delivery

After ten years of experience with iterative development projects, we have experience from 100s of projects to update our management principles. The transitional mix of disciplines promoted in iterative development needs to be updated to the more advanced economic disciplines of agile software delivery. What follows is my proposed top ten principles for achieving agile software delivery success.

Top 10 Management Principles of Agile Software Delivery

1. Reduce uncertainties by addressing architecturally significant decisions first.
2. Establish an adaptive life-cycle process that accelerates variance reduction.
3. Reduce the amount of custom development through asset reuse and middleware.

4. Instrument the process to measure cost of change, quality trends, and progress trends.
5. Communicate honest progressions and digressions with all stakeholders.
6. Collaborate regularly with stakeholders to renegotiate priorities, scope, resources, and plans.
7. Continuously integrate releases and test usage scenarios with evolving breadth and depth.
8. Establish a collaboration platform that enhances teamwork among potentially distributed teams.
9. Enhance the freedom to change plans, scope and code releases through automation.
10. Establish a governance model that guarantees creative freedoms to practitioners.

Successfully delivering software products in a predictable and profitable manner requires an evolving mixture of discovery, production, assessment, and a steering leadership style. The word "steering" implies active management involvement and frequent course-correction to produce better results. All stakeholders must collaborate to converge on moving targets, and the principles above delineate the economic foundations necessary to achieve good steering mechanisms. Three important conclusions that can be derived from these principles and practical experience are illustrated in Figure 4.

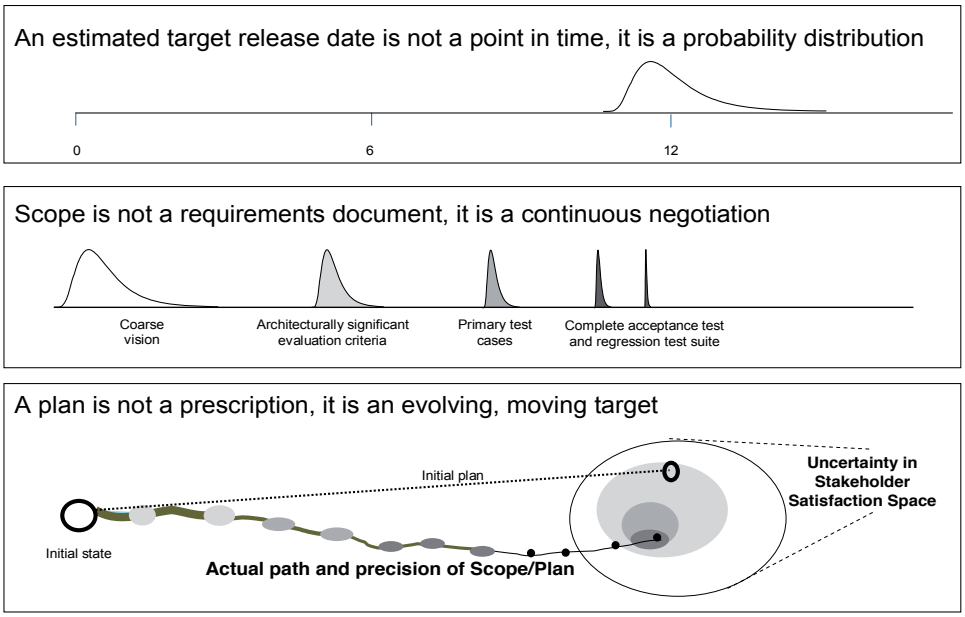


Figure 4: The governance of Agile software delivery means managing uncertainty and variance through steering

In a healthy software project, each phase of development produces an increased level of understanding in the evolving plans, specifications, and completed solution, because each phase furthers a sequence of executable capabilities and the team's knowledge of competing objectives. At any point in the life cycle, the precision of the subordinate artifacts should be in balance with the evolving precision in understanding, at compatible levels of detail and reasonably traceable to each other.

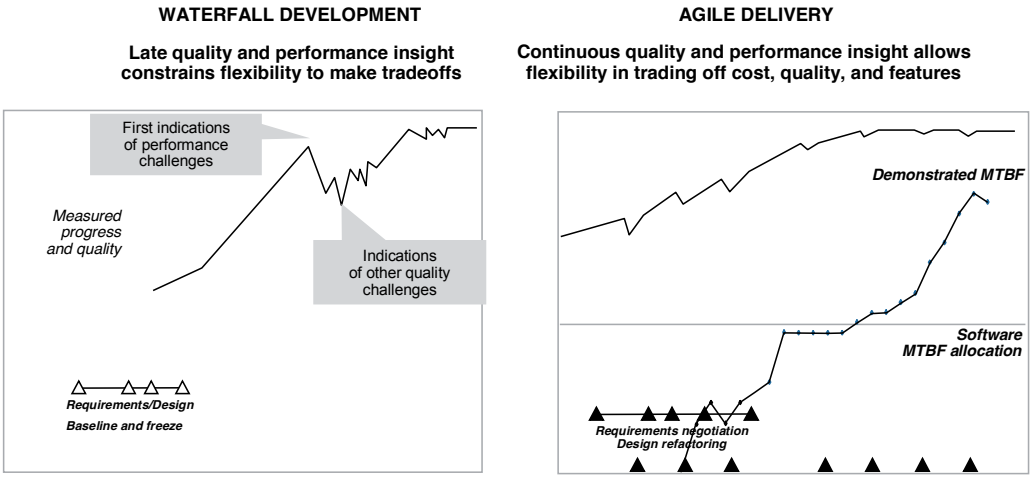
The difference between precision and accuracy in the context of software management is not trivial. Software management is full of gray areas, situation dependencies, and ambiguous tradeoffs. Understanding the difference between precision and accuracy is a fundamental skill of good software managers, who must accurately

forecast estimates, risks, and the effects of change. Precision implies repeatability or elimination of uncertainty. Unjustified precision — in requirements or plans — has proved to be a substantial yet subtle recurring obstacle to success. Most of the time, this early precision is just plain dishonest and serves to provide a counter-productive façade for portraying illusory progress and quality. Unfortunately, many sponsors and stakeholders demand this early precision and detail because it gives them (false) comfort of the progress achieved.

Iterative development processes have evolved into more successful agile delivery processes by improving the navigation through uncertainty with balanced precision. This steering requires dynamic controls and intermediate checkpoints, whereby stakeholders can assess what they have achieved so far, what perturbations they should make to the target objectives, and how to re-factor what they have achieved to adjust and deliver those targets in the most economical way. The key outcome of these modern agile delivery principles is increased flexibility, which enables the continuous negotiation of scope, plans, and solutions for effective economic governance.

Figure 5 provides another example of this important metric pattern. What this figure illustrates is the tangible evolution of a quality metric (in this case, the demonstrated mean time between failure for the software embedded in a large scale command and control system[14]). Whereas, the conventional process would have to deal speculatively with this critical performance requirement for most of the lifecycle, the project that employs a modern agile delivery approach eliminates the uncertainty in achieving this requirement early enough in the project's schedule that the team can effectively trade-off remaining resources to invest in more run-time performance, added functionality, or improved profit on

system delivery. This sort of reduction in uncertainty has significant economic leverage to all stakeholders.



- Speculative quality requirements
- Unpredictable cost/schedule performance
- Late shoehorning of suboptimal changes that impact quality
- Delays risk and uncertainty reduction until too late in the project life cycle

- Release qualities that matter
- Quality progressions/digressions
- Early requirement verification and/or negotiation
- Reduces critical sources of variance in cost to complete
- Increased flexibility in late resource investments

Figure 5: Reduced uncertainty in critical quality requirements improves the variance in the cost to complete and adds flexibility in downstream resource investments.

I have observed four discriminating patterns that are characteristic of successful agile delivery projects. These patterns represent a few "abstract gauges" that help the steering process to assess scope management, process management, progress management, and quality management. My hunch is that most project managers certified in traditional engineering project management will react negatively to these notions, because they run somewhat counter to conventional wisdom.

1. **Scope evolves:** Solutions evolve from stakeholder needs, and stakeholder needs evolve from available solutions assets. [Anti-pattern: Get all the requirements right up front.] This equal and opposite interaction between user need and solution is the engine for iteration that is driving more and more asset-based development. We just don't build many applications dominated by custom code development anymore. A vision statement evolves into interim evaluation criteria which evolve into test cases and finally detailed acceptance criteria. Scope evolves from abstract and accurate representations into precise and detailed representations as stakeholder understanding evolves (i.e., uncertainty is reduced).

2. **Process rigor evolves**: Process and instrumentation evolve from flexible to rigorous as the life-cycle activities evolve from early, creative tasks to later production tasks. [Anti-pattern: Define the entire project's life-cycle process as light or heavy.] Process rigor should be much like the force of gravity: the closer you are to a product release, the stronger the influence of process, tools, and instrumentation on the day-to-day activities of the workforce. The farther you are from a release date, the weaker the influence. This is a key requirement to be fulfilled by the development platform with automation support for process enactment if practitioners are to perceive a life-cycle process that delivers 'painless governance'.

3. **Progress assessment is honest**: Healthy projects display a sequence of progressions and digressions. [Anti-pattern: consistently progressing to 100% earned value as the original plan is executed, without any noticeable digression until late in the life cycle.] The transition to a demonstration-driven life cycle results in a very different project profile. Rather than a linear progression (often dishonest) of earned value, a healthy project will exhibit an honest sequence of progressions and digressions as they resolve

uncertainties, re-factor architectures and scope, and converge on an economically governed solution.

4. **Testing is the steering mechanism**: Testing of demonstrable releases is a full life-cycle activity and the cost of change in software releases improves or stabilizes over time. [Anti-pattern: testing is a subordinate, bureaucratic, late life-cycle activity and the cost of change increases over time]. Testing demands objective evaluation through *execution* of software releases under a controlled scenario with an expected outcome. In an agile delivery process that is risk-driven, integration testing will mostly precede unit testing and result in more flexibility in steering with more favorable cost of change trends.

With immature metrics and measures, software project managers are still overly focused on playing defense and struggling with subjective risk management. With further advances in software measurement and collaborative platforms that support process enactment of best practices and integrated metrics collection and reporting, we can manage uncertainty more objectively. Software project managers can invest more in playing offense through balancing risks with opportunities, and organizations can better exploit the value of software to deliver better economic results in their business.

A framework for reasoning about improving software economics

Today's empirical software cost estimation models (like COCOMO II, SEER, QSM Slim and others) allow users to estimate costs to within 25-30 percent, on three out of four projects[15]. This level of unpredictability in the outcome of software projects is a strong indication that software delivery and governance clearly requires

an economics discipline that can accommodate high levels of uncertainty. These cost models include dozens of parameters and techniques for estimating a wide variety of software development projects. For the purposes of this discussion, I will simplify these estimation models into a function of four basic parameters:

1. **Complexity**. The complexity of the software is typically quantified in units of human-generated stuff and its quality. Quantities may be assessed in lines of source code, function points, use-case points, or other measures. Qualities like performance, reuse, reliability, and feature richness are also captured in the complexity value. Simpler and more straightforward applications will result in a lower complexity value.

2. **Process**. This process exponent typically varies in the range 1.0 to 1.25 and characterizes the governance methods, techniques, maturity, appropriateness, and effectiveness in converging on wins for all stakeholders. Better processes will result in a lower exponent.

3. **Teamwork**. This parameter captures the skills, experience, motivations and know-how of the team along with its ability to collaborate toward well-understood and shared goals. More effective teams will result in a lower multiplier.

4. **Tools**. The tools parameter captures the extent of process automation, process enactment, instrumentation and team synchronization. Better tools will result in a lower multiplier.

The relationships among these parameters in modeling the estimated effort can be expressed as follows:

$$\text{Resources} = (\text{Complexity})^{(\text{Process})} * (\text{Teamwork}) * (\text{Tools})$$

By examining the mathematical form of this equation and the empirical data in the various models and their practical application across thousands of industry projects, one can easily demonstrate that these four parameters are in priority order when it comes to the potential economic leverage. In other words, a 10% reduction in complexity is worth more than a 10% improvement in the process, which is worth more than a 10% more capable team, which is worth more than a 10% increase in automation. In practice, this is exactly what IBM services teams have learned over the last twenty-five years of helping software organizations improve their software development and delivery capability.

We have been compiling best practices and economic improvement experiences for years. We are in the continuing process of synthesizing this experience into more consumable advice and valuable intellectual property in the form of value traceability trees, metrics patterns, benchmarks of performance, and instrumentation tools to provide a closed loop feedback control system for improved insight and management of the econometrics introduced earlier. Figure 6 summarizes the rough ranges of productivity impact and timeframes associated with many of the more common initiatives that IBM is investing in and delivering every day across the software industry. The impact on productivity typically affects only a subset of project and organization populations—they require savvy tailoring to put them into a specific context. As the scale of an organization grows, the impacts dampen predominantly because of standard inertia -- i.e., resistance to change.

We have been careful to present ranges and probability distributions to ensure that it is clear that "your mileage may vary." The key message from Figure 6 is that there is a range of incremental improvements that can be achieved and there is a

general hierarchy of impact. The more significant improvements, like systematic reduction in complexity and major process transformations, also require the more significant investments and time to implement. These tend to be broader organizational initiatives. The more incremental process improvements, skill improvements, and automation improvements targeted at individual teams, projects, or smaller organizations are more predictable and straightforward to deploy.

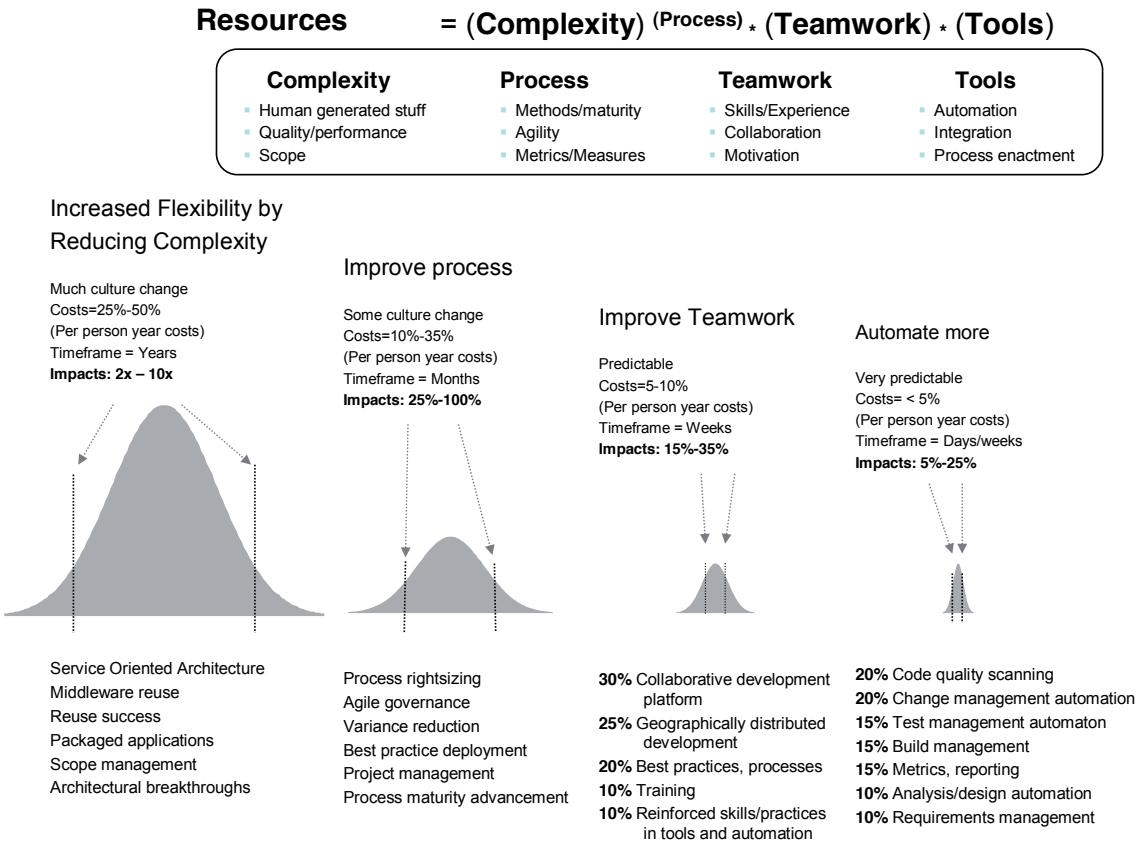


Figure 6: A rough overview of expected improvements for some best practices

The main conclusion that one can draw from the experience captured in Figure 6 is that ***improvements in each dimension have significant returns on investment***. The key to

substantial improvement in business performance is a balanced attack across the four basic parameters of the simplified software cost model: reduce complexity, streamline processes, optimize team contributions, and automate with tools. There are significant dependencies among these four dimensions of improvement. For example, new tools enable complexity reduction and process improvements; size reduction leads to process changes; collaborative platforms enable more effective teamwork; and process improvements drive tool advances. At IBM, and in our broad customer base of software development organizations, we have found that the key to achieving higher levels of improvements in teamwork, process improvement, and complexity reduction lies in supporting and reinforcing tooling and automation. Deploying best practices and changing cultures is more straightforward when you can systematically transform ways of working. This is done through deployment of tangible tools, which automate and streamline the best practices and are embraced by the practitioners, because these tools *increase* the practitioner's creative time spent in planning, analysis, prototyping, design, refactoring, coding, testing and deploying, while these tools *decrease* the time spent on unproductive activities such as unnecessary rework, change propagation, traceability, progress reporting, metrics collection, documentation, and training.

I realize that listing training among the unproductive activities will raise the eyebrows of some people. Training is an organizational responsibility, not a project responsibility. Any project manager who bears the burden of training people in processes, technologies, or tools is worse off than a project manager with a fully trained work force. A fully trained work force on every project is almost never possible, but employing trained people is always better than employing untrained people, other things being equal. In this sense, training is considered a non-value-added activity.

This is one of the fundamental dilemmas that organizations face as they try to improve in any one of the four dimensions. The overhead cost of training their teams on new things is a significant inhibitor to project success; this cost explains many managers' resistance to any new change initiative, whether it regards new tools, practices, or people.

In making the transition to new techniques and technologies, there is always apprehension and concern about failing, particularly by project managers who are asked to make significant changes in the face of tremendous uncertainty. Maintaining the status quo and relying on existing methods is usually considered the safest path. In the software industry, where most organizations succeed on less than half of their software projects, maintaining the status quo is not a safe bet. When an organization does decide to make a transition, two pieces of conventional wisdom are usually offered by both internal champions and external change agents: (1) Pioneer any new techniques on a small pilot program. (2) Be prepared to spend more resources – money and time – on the first project that makes the transition. In my experience, both of these recommendations are counterproductive.

Small pilot programs have their place, but they rarely achieve any paradigm shift within an organization. Trying out a new little technique, tool, or method on a very rapid, small-scale effort – less than three months, say, and with just a few people – can frequently show good results, initial momentum, or proof of concept. The problem with pilot programs is that they are almost never considered on the critical path of the organization. Consequently, they do not merit "A" players, adequate resources, or management attention. If a new method, tool, or technology is expected to have an adverse impact on the results of the

trailblazing project, that expectation is almost certain to come true. Why? Because software projects almost never do better than planned. Unless there is a very significant incentive to deliver early (which is very uncommon), projects will at best steer their way toward a target date. Therefore, the trailblazing project will be a non-critical project, staffed with non-critical personnel of whom less is expected. This adverse impact ends up being a self-fulfilling prophecy.

The most successful organizational paradigm shifts I have seen resulted from similar sets of circumstances: the organizations took their most critical project and highest caliber personnel, gave them adequate resources, and demanded better results on that first critical project.

Conclusion

Day-to-day decisions in software projects have always been, and continue to be, dominated by decisions rooted in the tradition of economics discipline, namely: value judgments, cost tradeoffs, human factors, macro-economic trends, technology trends, market circumstances, and timing. Software project activities are rarely concerned with engineering disciplines such as mathematics, material properties, laws of physics, or established and mature engineering models. The primary difference between economics and engineering governance is the amount of uncertainty inherent in the product under development. The honest treatment of uncertainty is the foundation of today's best practices; we have learned over and over that what makes a software practice better or best is that the practice reduces uncertainty in the target outcome.

Here are four concluding thoughts that summarize the main themes of this paper:

1. Agile software delivery is better served by economic governance principles. With software delivery becoming a more dominant business process in most product, systems, and services companies, the predictability and track record of applying conventional engineering principles to managing software won't be competitive.

2. Our top ten principles of agile software delivery have a common theme: They describe "economic governance" approaches that attack uncertainties and reduce the variance in the estimate to complete.

3. The primary metric for demonstrating that an organization or project has transitioned to effective agile delivery is the trend in the cost of change. This measure of the adaptability inherent in software releases is a key indicator of the flexibility required to continuously navigate uncertainties and steer projects toward success.

4. The next wave of technological advances to improve the predictability and outcomes of software economics needs to be in measurement and instrumentation that supports better economic governance.

   IBM, and the Rational organization in particular, will continue to invest in research, practices, measures, instrumentation, and tools to advance our knowledge and practice of software economic governance, so that our customers can exploit a mature business process for agile software delivery.

References

1. Royce, Bittner, Perrow, *The Economics of Software Development*, Addison-Wesley, 2009.
2. Royce, Walker, "Successful Software Management Style: Steering and Balance," IEEE Software, Vol. 22, No. 5, September/October 2005
3. Royce, Winston W., "Managing the Development of Large Software Systems," IEEE Wescon, 1970.
4. Kruchten, Philippe, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999, 2003.
5. Kruchten, Philippe, Kroll, Per, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison-Wesley, 2003.
6. Hubbard, Douglass W., *How to Measure Anything*, John Wiley and Sons, 2007.
7. Williams, Laurie, Kessler, Robert. *Pair Programming Illuminated*, Addison Wesley, 2003.
8. Williams, L., Krebs, W., Layman, L., Anton, A., "Toward a Framework for Evaluating Extreme Programming, Empirical Assessment in Software Engineering," (EASE), 2004.
9. Austin, Robert, Devin, Lee, *Artful Making*, FT Press, 2003.

Endnotes

1 Royce, Bittner, Perrow, *The Economics of Software Development*, Addison-Wesley, 2009.

2 Royce, Walker, "Successful Software Management Style: Steering and Balance," IEEE Software, Vol. 22, No. 5, September/October 2005

3 Williams, Laurie, Kessler, Robert. *Pair Programming Illuminated*, Addison Wesley, 2003.

4 Williams, L., Krebs, W., Layman, L., Anton, A., "Toward a Framework for Evaluating Extreme Programming, Empirical Assessment in Software Engineering," (EASE), 2004.

5 Royce, Walker E., *Software Project Management,* Addison Wesley, 1998.

6 Royce, Winston W., "Managing the Development of Large Software Systems," *IEEE Wescon*, 1970.

7 Royce, Op cit.

8 Royce, Op cit.

9 Royce, Op cit.

10 Appendix D in my book *Software Project Management* provides a large scale case study of a DoD project that achieved the cost of change pattern on the right side of Figure 2.

11 Kruchten, Philippe, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999, 2003.

12 The variance of a random variable (i.e., a probability distribution or sample) is a measure of statistical dispersion. Technically, variance is defined as the average of the squared distance of all values from the mean. The mean describes the expected value and the variance represents a measure of uncertainty in that expectation. The square root of the variance is called the standard deviation and is a more accepted measure since it has the same units as the random variable.

13 Hubbard, Douglass W., *How to Measure Anything*, John Wiley and sons, 2007.

14 Royce, Op cit.

15 Boehm, Barry. *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, 2000.

IBM Corporation

Software Group

Route 100

Somers, NY 10589

U.S.A.

Produced in the United States of America

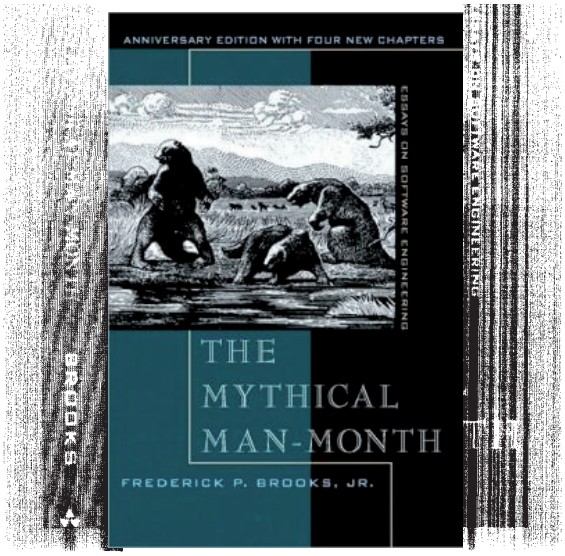
May 2009

Artikel 4 bij leereenheid 2

The Mythical Man-Month, after 20 Years
Frederick P. Brooks

IEEE Software, volume 12, issue 5, March 2012

BOOK EXCERPT



The Mythical Man-Month

AFTER 20 YEARS

Frederick P. Brooks, Jr.

Excerpted from Chapter 19, "The Mythical Man-Month After 20 Years," in The Mythical Man-Month: Anniversary Edition, *copyright © 1995 by Addison-Wesley Publishing Company, Inc. Reprinted with permission.*

Artwork courtesy The George C. Page Museum of La Brea Discoveries, The Natural History Museum of Los Angeles County.

WHY IS THERE A TWENTIETH ANNIVERSARY EDITION?

The plane droned through the night toward LaGuardia. Clouds and darkness veiled all interesting sights. The document I was studying was pedestrian. I was not, however, bored. The stranger sitting next to me was reading *The Mythical Man-Month*, and I was waiting to see if by word or sign he would react. Finally as we taxied toward the gate, I could wait no longer:

"How is that book? Do you recommend it?"

"Hmph! Nothing in it I didn't know already."

I decided not to introduce myself.

Why has *The Mythical Man-Month* persisted? Why is it still seen to be relevant to software practice today? Why does it have a readership outside the software engineering community, generating reviews, citations, and correspondence from lawyers, doctors, psychologists, sociologists, as well as from software people? How can a book written 20 years ago about a software-building experience 30 years ago still be relevant, much less useful?

One explanation sometimes heard is that the software development discipline has not advanced normally or properly. This view is often supported by contrasting computer software development productivity with computer hardware manufacturing productivity, which has multiplied at least a thousandfold over the two decades. As Chapter 16 explains, the anomaly is not that software has been so slow in its progress but rather that computer technology has exploded in a fashion unmatched in human history. By and large this comes from the gradual transition of computer man-

ufacturing from an assembly industry to a process industry, from labor-intensive to capital-intensive manufacturing. Hardware and software development, in contrast to manufacturing, remain inherently labor-intensive.

A second explanation often advanced is that *The Mythical Man-Month* is only incidentally about software but primarily about how people in teams make things. There is surely some truth in this; in the preface to the 1975 edition I said that managing a software project is more like other management than most programmers initially believe. I still believe that to be true. Human history is a drama in which the stories stay the same, the scripts of those stories change slowly with evolving cultures, and the stage settings change all the time. So it is that we see our twentieth-century selves mirrored in Shakespeare, Homer, and the Bible. So to the extent that *The MM-M* is about people and teams, obsolescence should be slow.

Whatever the reason, readers continue to buy the book, and they continue to send me much-appreciated comments. Nowadays I am often asked, "What do you think was wrong when written? What is now obsolete? What is really new in the software engineering world?" These quite distinct questions are all fair, and I shall address them as best I can. Not in that order, however, but in clusters of topics.

PARNAS WAS RIGHT AND I WAS WRONG ABOUT INFORMATION HIDING

In Chapter 7 I contrast two approaches to the question of how much each team member should be allowed or encouraged to know about the designs and code of other team members. In the Operating System/360 project, we decided that *all* programmers should see *all* material, i.e., each programmer having a copy of the project workbook, which came to number over 10,000 pages. Harlan Mills has argued persuasively that "programming should be a public process," that exposing all the work to everybody's gaze helps quality control both by peer pressure to do things well and by peers actually spotting flaws and bugs.

This view contrasts sharply with David Parnas's teaching that modules of code should be encapsulated with well-defined interfaces, and that the interior of such a module should be the private property of its programmer, not discernible from outside. Programmers are most effective if shielded from, not exposed to, the innards of modules not their own.

I dismissed Parnas's concept as a "recipe for disaster" in Chapter 7. Parnas was right, and I was wrong. I am now convinced that information hiding, today often embodied in object-oriented programming, is the only way of raising the level of software design.

One can indeed get disasters with either technique. Mills' technique ensures that programmers can know the detailed semantics of the interfaces they work to by knowing what is on the other side. Hiding those semantics leads to system bugs. On the other hand, Parnas's technique is robust under change and is more appropriate in a design-for-change philosophy. Chapter 16 argues the following:

- Most past progress in software productivity has come from eliminating noninherent difficulties such as awkward machine languages and slow batch turnaround.
- There are not a lot more of these easy pickings.
- Radical progress is going to have to come from attacking the essential difficulties of fashioning complex conceptual constructs.

The most obvious way to do this recognizes that programs are made up of conceptual chunks much larger than the individual high-level language statement — subroutines, or modules, or classes. If we can limit design and building so that we only do the putting together and parameterization of such chunks from prebuilt collections, we have radically raised the conceptual level, and eliminated the vast amounts of work and the copious opportunities for error that dwell at the individual statement level.

Parnas's information-hiding definition of modules is the first published step in that crucially important research program, and it is an intellectual ancestor of object-oriented programming. He defined a module as a software entity with its own data model and its own set of operations. Its data can only be accessed via one of its proper operations. The second step was a contribution of several thinkers: the upgrading of the Parnas module into an *abstract data type*, from which many objects could be derived. The abstract data type provides a uniform way of thinking about and specifying module interfaces, and an access discipline that is easy to enforce.

The third step, object-oriented programming, introduces the powerful concept of *inheritance*, whereby classes (data types) take as

defaults specified attributes from their ancestors in the class hierarchy. Most of what we hope to gain from object-oriented programming derives in fact from the first step, module encapsulation, plus the idea of prebuilt libraries of modules or classes *that are designed and tested for reuse.* Many people have chosen to ignore the fact that such modules are not just programs, but instead are program products in the sense discussed in Chapter 1. Some people are vainly hoping for significant module reuse without paying the initial cost of building product-quality modules — generalized, robust, tested, and documented. Object-oriented programming and reuse are discussed in Chapters 16 and 17.

PEOPLE ARE EVERYTHING (WELL, ALMOST EVERYTHING)

Some readers have found it curious that *The MM-M* devotes most of the essays to the managerial aspects of software engineering, rather than the many technical issues. This bias was due in part to the nature of my role on the IBM Operating System/360 (now MVS/370). More fundamentally, it sprang from a conviction that the quality of the people on a project, and their organization and management, are much more important factors in success than are the tools they use or the technical approaches they take.

Subsequent researches have supported that conviction. Boehm's COCOMO model finds that the quality of the team is by far the largest factor in its success, indeed four times more potent than the next largest factor. Most academic research on software engineering has concentrated on tool. I admire and covet sharp tools. Nevertheless, it is encouraging to see ongoing research efforts on the care, growing, and feeding of people, and on the dynamics of software management.

Peopleware. A major contribution during recent years has been DeMarco and Lister's 1987 book, *Peopleware: Productive Projects and Teams.* Its underlying thesis is that "The major problems of our work are not so much *technological* as *sociological* in nature." It abounds with gems such as, "The manager's function is not to make people work, it is to make it possible for people to work." It deals with such mundane topics as space, furniture, team meals together. DeMarco and Lister provide real data from their Coding War Games that show stunning correlation between performances of programmers from the same organization, and between workplace characteristics and both productivity and defect levels.

The top performers' space is quieter, more private, better protected against interruption, and there is more of it.... Does it really matter to you ... whether quiet, space, and privacy help your current people to do better work or [alternatively] help you to attract and keep better people?

I heartily recommend the book to all my readers.

Moving projects. DeMarco and Lister give considerable attention to team *fusion*, an intangible but vital property. I think it is management's overlooking fusion that accounts for the readiness I have observed in multilocation companies to move a project from one laboratory to another.

My experience and observation are limited to perhaps a half-dozen moves. I have never seen a successful one. One can move *missions* successfully. But in every case of attempts to move projects, the new team in fact started over, in spite of having good documentation, some well-advanced designs, and some of the people from the sending team. I think it is the breaking of fusion of the old team that aborts the embryonic product, and brings about restart.

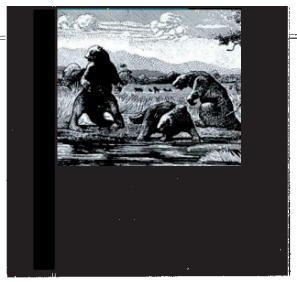
THE POWER OF GIVING UP POWER

If one believes, as I have argued at many places in this book, that creativity comes from individuals and not from structures or processes, then a central question facing the software manager is how to design structure and process so as to enhance, rather than inhibit, creativity and initiative. Fortunately, this problem is not peculiar to software organizations, and great thinkers have worked on it. E.F. Schumacher, in his classic, *Small is Beautiful: Economics as if People Mattered,* proposes a theory of organizing enterprises to maximize the creativity and joy of the workers. For his first principle he chooses the "Principle of Subsidiary Function" from the Encyclical *Quadragesimo Anno* of Pope Leo XIII:

It is an injustice and at the same time a grave evil and disturbance of right order to assign to a greater and higher association what lesser and subordinate organizations can do. For every social activity ought of its very nature to furnish help to the members of the body social and never destroy and absorb them.... Those in command should be sure that the more perfectly a graduated order is preserved among the various associations, in observing the principle of subsidiary function, the stronger will be the social authority and effectiveness and the happier and more prosperous the condition of the State.

Schumacher goes on to interpret:

The Principle of Subsidiary Function teaches us that the centre will gain in authority and effectiveness if the freedom



and responsibility of the lower formations are carefully preserved, with the result that the organization as a whole will be "happier and more prosperous."

How can such a structure be achieved? ... The large organization will consist of many semi-autonomous units, which we may call quasi-firms. *Each of them will have a large amount of freedom, to give the greatest possible chance to creativity and* entrepreneurship.... *Each quasi-firm must have both a profit and loss account, and a balance sheet.*

Among the most exciting developments in software engineering are the early stages of putting such organizational ideas into practice. First, the microcomputer revolution created a new software industry of hundreds of start-ups. These firms, all of them starting small, and marked by enthusiasm, freedom, and creativity. The industry is changing now, as many small companies continue to be acquired by larger ones. It remains to be seen if the larger acquirers will understand the importance of preserving the creativity of smallness.

More remarkably, high management in some large firms have undertaken to delegate power down to individual software project teams, making them approach Schumacher's quasi-firms in structure and responsibility. They are surprised and delighted at the results.

Jim McCarthy of Microsoft described to me his experience at emancipating his teams:

Each feature team (30-40 people) owns its feature set, its schedule, and even its process of how to define, build, ship. The team is made up for four or five specialties, including building, testing, and writing. The team settles squabbles; the bosses don't. I can't emphasize enough the importance of empowerment, of the team being accountable to itself for its success.

Earl Wheeler, retired head of IBM's software business, told me his experience in undertaking the downward delegation of power long centralized in IBM's division managements:

The key thrust [of recent years] was delegating power down. It was like magic! Improved quality, productivity, morale. We have small teams, with no central control. The teams own the process, but they have to have one. They have many different processes. They own the schedule, but they feel the pressure of the market. This pressure causes them to reach for tools on their own.

Conversations with individual team members, of course, show both an appreciation of the power and freedom that is delegated, and a somewhat more conservative estimate of how much control really is relinquished. Nevertheless, the delegation achieved is clearly a step in the right direction. It yields exactly the benefits Leo XIII predicted: the center gains in real authority by delegating power, and the organization as a whole is happier and more prosperous. ◆

Artikel 5 bij leereenheid 2

The Profession of IT, Evolutionary System Development
Peter J. Denning, Chris Gunderson en Rick Hayes-Roth

Communications of the ACM, volume 51, issue 12, December 2008

DOI:10.1145/1409360.1409371 Peter J. Denning, Chris Gunderson, and Rick Hayes-Roth

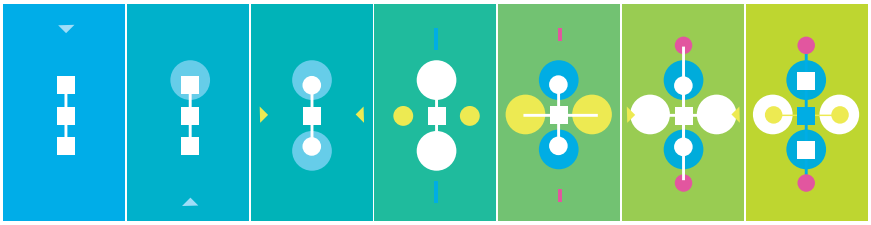
The Profession of IT
Evolutionary System Development

Large systems projects are failing at an alarming rate. It's time to take evolutionary design methods off the shelf.

Many critical large systems are failing. The replacement FAA air traffic control system, the FBI virtual case file, and the Navy Marine Corps Internet (NMCI), are a few of the many billion-dollar systems that could not deliver the functions needed. In stark contrast, the Boeing 777 aircraft, the Global Positioning System (GPS), and the U.S. Census database system have been outstanding successes. Why do some systems fail and others succeed?

Development time is the critical factor. This is the time to deliver a system that meets the requirements set at the beginning of the development process. If development time is shorter than the environment change time, the delivered system is likely to satisfy its customers. If, however, the development time is long compared to the environment change time, the delivered system becomes obsolete, and perhaps unusable, before it is finished. In government and large organizations, the bureaucratic acquisition process for large systems can often take a decade or more, whereas the using environments often change significantly in as little as 18 months (Moore's Law).

The Boeing 777, GPS, and U.S. Census data systems were developed for stable environments—they were completed before any significant changes occurred in their requirements. In contrast, the FAA replacement system, FBI Virtual Case File (see www.spectrum.ieee.org/sep05/1455), NMCI (GAO[4], www.nmcistinks.com) all faced dynamic environments that changed faster than their development processes could. Predecessors of these systems were successful because their environments were stable, but the current generations encountered trouble because their environments had become too dynamic.



The traditional acquisition process tries to avoid risk and control costs by careful preplanning, anticipation, and analysis. For complex systems, this process usually takes a decade or more. Are there any alternatives that would take much less time and still be fit for use?

Yes. Evolutionary system development produces large systems within dynamic social networks. The Internet, World Wide Web, and Linux are prominent examples. These successes had no central, preplanning process, only a general notion of the system's architecture, which provided a framework for cooperative innovation. Individuals in the network banded into small groups to quickly produce or modify modules in the architecture. They tested their modules by asking other users to try them. The systems evolved rapidly in many small increments that aligned with current perceptions of the using environment.

Moreover, the evolutionary process embraces risk, and the patience to see what emerges. It works with nature's principle of fitness in the environment: components that work well survive, and those that do not are abandoned.

The astonishing success of evolutionary development challenges our common sense about developing large systems. We need to learn from these systems, because evolutionary development may be the only way to achieve satisfactory replacements for aging large systems and to create new, unprecedented systems.

Evolutionary development is a mature idea that has languished away from mainstream practice. In this column, we will analyze why evolutionary development does not fit the current common sense and why we need to work to change that.

Our Current Common Sense

From its founding in 1968, the software engineering field set out to address the "software crisis," a persistent inabil-

ity to deliver dependable and usable software. Fritz Bauer, one of the field's founders, believed a rigorous engineering approach was needed. He famously quipped, "Software engineering is the part of computer science that is too hard for computer scientists." Over the years, software engineers produced many powerful tools: languages, module managers, version trackers, visualizers, and debuggers are some examples. In his famous "No silver bullet" assessment (1986), Fred Brooks concluded that the software crisis had not abated despite huge advancements in tools and methods; the real problem was getting an intellectual grasp of the problem and translating that understanding into an appropriate system architecture.[2] The tools of 1986, while better than those of 1968, relied on concepts that did not scale up to ever-larger systems. The situation today is much the same: tools are more powerful, but we struggle with scalability, usability, and predictability.

Current software engineering is based on four key assumptions:

- Dependable large systems can only be attained through rigorous application of the engineering design process (requirements, specifications, prototypes, testing, acceptance).
- The key design objective is an architecture that meets specifications derived from knowable and collectable requirements.
- Individuals of sufficient talent and experience can achieve an intellectual grasp of the system.
- The implementation can be completed before the environment changes very much.

What if these assumptions no longer hold? The first assumption is challenged by the failures of large systems that used the traditional design process and the successes of other large systems that simply evolved. The remaining assumptions are challenged by the increasingly dynamic environments, often called ecosystems, in which large systems operate. There is no complete statement of requirements because no one person, or even small group, can have complete knowledge of the whole system or can fully anticipate how the community's requirements will evolve.

The astonishing success of evolutionary development challenges our common sense about developing large systems.

System Evolution: A New Common Sense

To avoid obsolescence, therefore, a system should undergo continual adaptation to the environment. There are two main alternatives for creating such adaptations. The first, successive releases of a system, is the familiar process of software product releases. It can work in a dynamic environment only when the release cycle is very short, a difficult objective under a carefully prescribed and tightly managed process. Windows Vista, advertised as an incremental improvement over XP, was delivered years late and with many bugs.

The second approach to adaptation is many systems competing by mimicking natural evolution; the more fit systems live on and the less fit die out. Linux, the Internet, and the World Wide Web illustrate this with a constant churn of experimental modules and subsystems, the best of which are widely adopted.

Evolutionary system design can become a new common sense that could enable us to build large critical systems successfully. Evolutionary approaches deliver value incrementally. They continually refine earlier successes to deliver more value. The chain of increasing value sustains successful systems through multiple short generations.

Designs by Bureaucratic Organizations

Fred Brooks observed that software tends to resemble the organization that built it. Bureaucratic organizations tend toward detailed processes constrained by many rules. The U.S. government's standard acquisition practices, based on careful preplanning and risk avoidance, fit this paradigm. Their elaborate architectures and lengthy implementation cycles cannot keep up with real, dynamic environments.

It may come as a surprise, therefore, that practices for adaptability are allowed under government acquisition rules. In 2004, the Office of Secretary of Defense sponsored the launch of W2COG, the World Wide Consortium for the Grid (w2cog.org) to help advance networking technology for defense using open-development processes such as in the World Wide Web Consortium (w3c.org). The W2COG took advantage of a provision of acquisition regulations that allows Limited Technology Experiments (LTEs). The W2COG recently completed an experiment to develop a secure service-oriented architecture system, comparing an LTE using evolutionary methods against a standard acquisition process. Both received the same government-furnished software for an initial baseline. Eighteen months later, the LTE's process delivered a prototype open architecture that addressed 80% of the government requirements, at a cost of $100K, with all embedded software current, and a plan to transition to full COTS software within six months.

In contrast, after 18 months, the standard process delivered only a concept document that did not provide a functional architecture, had no working prototype, deployment plan, or timeline, and cost $1.5M. The agile method produced a "good enough" immediately usable 80% success for 1/15 the cost of the standard method, which seemed embarked on the typically long road to disappointment.

Agile Methods for Large Systems

Agile system development methods have been emerging for a decade.[1,3,6] These methods replace the drawn-out preplanning of detailed specifications with a fast, cyclic process of prototyping and customer interaction. The evolutionary design approach advocated here is a type of agile process.

The U.S. Government Accounting Office (GAO) has scolded the government on several occasions for its uncommitted lip service to agile processes.[4] The GAO believes agile processes could significantly shorten time to delivery, reduce failure rate, and lower costs. Many people resist the GAO advice because

they assume careful preplanning minimizes risk and maximizes dependability and usability. However, more leaders are pushing for agile acquisition because the track record of the normal process in dynamic environments is so dismal.

The software engineering community has hotly debated preplanned versus agile processes. After a while they reached a truce where they agreed that preplanning is best for large systems where reliability and risk-avoidance are prime concerns, and agile is best for small to medium systems where adaptability and user friendliness are prime concerns.

We challenge that conclusion. Preplanning is ceasing to be a viable option for large systems. Moreover, many small systems aim to be ultra-reliable.

Evolutionary Ecosystems

Evolutionary development uses "loosely managed" processes. Numerous successful large systems evolved through such a process—CTSS, Unix, Linux, Internet, Google, Amazon, eBay, Apple iPhone Apps, and banking applications are notable examples. All these systems relied on a common platform used by all members of the community, from developers to users. In such an ecosystem, successful prototypes transition easily to working products. It appears that the common ecosystem provides enough constraints that loose management works. The successful ecosystems were guided by a vision and a set of interaction rules that everyone in the community accepted. Building ecosystems for governments is quite challenging because of organizational impediments to information sharing.[5] We advocate much more aggressive use of loosely managed ecosystems. The W2COG was conceived to allow government to join a large ecosystem that could adaptively address its information networking needs.

Loosely managed does not mean unmanaged. Scrum and Extreme Programming (XP) are often cited as successful management approaches for agile processes.[6] Even the respected Capability Management Model (CMM) is amenable to agile development.

Whereas preplanned development seeks to avoid risks, evolutionary development mimics nature and embraces risks. The developers purposely expose emerging systems to risks to see how they fail, and then they build better system variants. It is better to seek risk out and learn how to survive it. In a natural ecosystem, only the most fit organisms survive. Fitness is nature's way of managing risk.

Whereas preplanned development seeks to avoid risks, evolutionary development mimics nature and embraces risks.

All the evidence says that that evolutionary processes works for systems large and small, and that risk seeking is the fastest route to fitness. There is too much at stake to continue to allow us to be locked into a process that does not work. C

References

1. Boehm, B. Making a difference in the software century. *IEEE Computer* (Mar. 2008), 32–38.
2. Brooks, F. *The Mythical Man Month.* Anniversary Edition. Addison-Wesley, 1995.
3. Cao, L. and Balascubramaniam, R. Agile software development: Ad hoc practice or sound principles? *IEEE Pro* (Mar.–Apr. 2007), 41–47.
4. GAO. *Defense Acquisitions: Assessments of Selected Weapons Programs.* Report GAO-06-391 (Mar. 2006); http://www.gao.gov/new.items/d06391.pdf, and *Information Technology: DOD Needs to Ensure That Navy Marine Corps Intranet Program Is Meeting Goals and Satisfying Customers.* Report GAO-07-51. (Dec. 2006); http://www.gao.gov/new.items/d0751.pdf.
5. Hayes-Roth, R., Blais, C., Brutzman, D. and Pullen, M. How to implement national information sharing strategy. *AFCEA-GMU C4I Center Symposium: Critical Issues in C4I*, George Mason University, Fairfax, VA, AFCEA (2008); http://c4i.gmu.edu/events/reviews/2008/papers/25_Hayes-Roth.pdf.
6. Schwaber, K. *Agile Project Management with Scrum.* Microsoft Press, 2004.

Peter J. Denning (pjd@nps.edu) is the director of the Cebrowski Institute for Information Innovation and Superiority at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.

Chris Gunderson (cgunders@w2cog.org), Captain (retired) U.S. Navy, is Principal Investigator of the Naval Postgraduate School W2COG and Netcentric Certification Office initiatives.

Rick Hayes-Roth (hayes-roth@nps.edu) is Professor of Information Systems at the Naval Postgraduate School in Monterey, California, and was CTO for Software at Hewlett-Packard Company.

Artikel 6 bij leereenheid 2

In-House Software Development: What Project Management Practices Lead to Success?
June M. Verner en W.M. Evanco

IEEE Software, volume 22, no. 1, January/February 2005

In-House Software Development: What Project Management Practices Lead to Success?

June M. Verner, *National Information and Communications Technology Australia*

William M. Evanco, *Drexel University*

Project management is an important part of software development, both for organizations that rely on third-party software development and for those whose software is developed primarily in-house. However, most software engineering research emphasizes "technical matters above behavioral matters."[1] Moreover, quantitative survey-based research regarding software development's early, nontechnical aspects is lacking. Additionally, in-house software development failures are unlikely to receive the same attention as third-party software development failures with their attendant litigation and negative media coverage.

To help provide a project management perspective for managers responsible for in-house software development, we conducted a survey in an attempt to determine the factors that lead to successful projects. We chose a survey because of its simplicity and because we hoped to find relationships among variables. Also, a survey let us cover more projects at a lower cost than would an equivalent number of interviews or a series of case studies. Our results provide general guidance for business and project managers to help ensure that their projects succeed.

A survey of in-house software development practices investigated why projects succeed or fail. A clear vision of the final product, good requirements, active risk management, and postmortem reviews can all help increase the odds of success.

Our questionnaire

We conducted wide-ranging, structured discussions with 21 senior software practitioners at a large financial organization to document their views regarding software project success or failure and the practices they consider important. We developed our questionnaire on the basis of these discussions, which focused on the practitioners' recent projects.

This practitioner group responded to our questionnaire twice—for a project they considered successful and for one they considered a failure (42 projects total). Later, we distributed our questionnaire to practitioners who developed in-house software for a variety of

commercial US organizations such as financial institutions, banks, pharmaceutical companies, and insurance companies. This set of questionnaires included descriptions of 80 unique projects. In total, we surveyed 101 respondents about 122 projects. Our sample wasn't random, but rather a convenience sample of practitioners we know.

We organized the questionnaire into seven sections, three of which we discuss in this article: project management, requirements elicitation and management, and cost and effort estimation and scheduling. We also asked respondents if they considered the referenced project successful (according to their own definitions of success). We discuss the questionnaire's remaining four sections elsewhere.[2]

Results and analysis

Our sample size (standard error 0.002) was reasonably large for empirical software engineering research. Of all the projects in the survey, the respondents regarded 62 percent as successful and 38 percent as unsuccessful. Eighty-seven percent were development projects (55 percent successful), and 13 percent were large (in terms of effort) maintenance or enhancement projects (60 percent successful). Overall, 64 percent of our projects had nine or fewer full-time employees, 27 percent had between 10 and 19, and the rest had 20 or more, with a median of eight.

Nineteen percent of the projects had no prescribed development methodology. Fifty-seven percent used a waterfall methodology, five percent a modified waterfall, five percent prototyping, two percent incremental delivery, two percent a spiral model, and 10 percent an in-house proprietary methodology.

Table 1 shows the percentage of yes responses to the survey questions. It also shows significant correlations with project success (<0.05) as well as some associations between responses to selected questions. The table shows five aspects of project management: the project manager (prefixed with M in column one), requirements analysis (R), cost and effort estimation and scheduling (C), risk assessment (A), and postmortem (P).

Project management

Because we assume that project management and the project manager (PM) are pivotal to a project's outcome, we begin by analyzing PM participation in the development process. Although you'd expect software development projects to have PMs, five percent of our sample projects didn't. Most of these projects were small, with fewer than seven full-time personnel equivalents. However, one failed project with 100 internal practitioners and 25 contractors had no PM. In 16 percent of the projects, the PM changed at least once. This volatility, practitioners reported, was very disruptive. The largest project in our survey had 80 internal practitioners and 100 contractors, and the PM was changed; the practitioners viewed it as a failure. For all projects, changing the PM was significantly negatively correlated with project success.

For all projects, changing the project manager was significantly negatively correlated with project success.

The PMs' software development experience ranged from under six months to 22 years, with a median of five years. Fifty-one percent of PMs had a software development background, 35 percent had a business background, and the rest had other backgrounds, such as engineering. Neither a PM with a software development background (M14) nor one experienced in the application area (M2) was significantly associated with project success. This result agrees with observations that a broad background is more useful than expertise in any particular technical area. According to Jaak Jurison, "[s]uccessful project managers are generalists, not technical specialists." Although some technical competence is helpful, managerial and interpersonal skills are more important.[3] Our results support this perspective; we found that the abilities to communicate (M5) and relate well with staff (M6) were significantly associated and positively correlated with success (this type of relationship is referred to as "positively associated" from here on).

An above-average PM (M1) was also positively associated with project success. This isn't surprising, because "poor management can increase software costs more rapidly than any other factor."[4] Table 1 also shows that an above-average PM (M1) is involved with good schedule estimates (C4) made with appropriate requirements information (C2). A PM with a clear vision of the project (M4) was also positively associated with success. Lack of a clear vision leads to poorly defined goals and specifications, poor requirements, insufficient project-planning time, lack of a project plan, and unrealistic deadlines and budgets.[3]

Table 1

Survey results

ID	Question	Yes answers (%)			Direction of success relationship	χ^2 association with project success	Association with other questions*†
		Successful projects	Failures	All projects			
M0	Did the project have a project manager (PM)?	97	95	96	**	**	**
M1	Was the PM above average?	76	34	55	+	0.000	M2, M3, M4, M5, M7, M8, R1, R2, R4, R5, R7, R8, C2, C3, C4, C5(–), C6, C7, P1
M2	Was the PM experienced in the application area?	68	73	62	**	**	M3, M5
M3	Did the PM understand the customer's problems?	79	49	67	+	0.001	M7, R1, R4, R5, R7, R9, R10(–), C4, C6
M4	Did the PM have a clear vision of the project?	86	44	71	+	0.000	M4, M7, R2, R4, R5, R7, C2, C4
M5	Did the PM communicate well with the staff?	68	32	54	+	0.000	M3, M4, M5, M7, R1, R2, R4, R5, R7, R8, C2, C4, C5(–), C6, C7
M6	Did the PM relate well with the development staff?	70	35	57	+	0.001	**
M7	Did the PM delegate authority?	94	83	90	**	**	R4, R9, C2, C7
M8	Did the PM pitch in and help when necessary?	56	61	58	**	**	M2, M3, M5, R2(–), C1(–), C5, C6(–))
M9	Did the PM control the project?	74	56	68	+	0.042	M1, C5(–), C6
M10	Did the PM treat the staff equally?	78	56	70	+	0.020	M1, M2, M3, M4, M9, R2, R4, R5, R7, R8, C2, C3, C5(–)
M11	Did the PM let the staff know he/she appreciated their working long hours?	74	51	65	+	0.008	M1, M3, M4, M5, M10, R7, C2, C3, C4, C5(–), C7(–)
M12	Did the PM ensure the staff was rewarded for working long hours?	48	21	37	+	0.003	M1, M3, M4, M5, M9, M10, M11, R1, R2, R5, C2, C3, C4
M13	Was the PM changed during the project?	11	24	16	–	0.05	M4(–), M8(–), R2(–), A2(–)
M14	Did the PM have a software development background?	49	54	51	**	**	**
R1	Were requirements gathered using a specific method?	50	49	50	**	**	R2, R8
R2	Were requirements complete and accurate at the project's start?	45	19	52	+	0.004	R1, R6
R3	If not complete at start, were the requirements completed later?	76	26	53	+	0.000	**

** All associations are reciprocal but are only shown once.*
*** There is no relationship or the relationship is shown as its reciprocal.*
† A negative symbol in parentheses means variables are significantly associated with a negative correlation.

Our data supports the view that management support of the development team is essential to motivate the team to work effectively toward organizational goals.[5] Appreciating and rewarding staff who worked long hours (M11, M12) were positively associated with success.

Using logistic regression (we required in our logistic regressions that the explanatory variable have at least a five percent level of significance), we found that the best PM predic-

Table 1

Survey results (cont.)

ID	Question	Yes answers (%)			Direction of success relationship	χ^2 association with project success	Association with other questions*†
		Successful projects	Failures	All projects			
R4	Overall, were the requirements good?	86	39	68	+	0.000	R2
R5	Did the project have a well-defined scope?	79	40	64	+	0.000	R2, R4
R6	Did the scope increase during the project?	61	74	66	**	**	R2(–), R9(–), R10
R7	Did users make adequate time available for requirements gathering?	83	40	66	+	0.000	R1, R2, R4, R6(–), R8
R8	Was there a central repository for requirements?	73	42	61	+	0.003	R4, R5
R9	Did the requirements result in well-defined deliverables?	76	35	60	+	0.000	R1, R2, R4, R5, R8
R10	Did the project's size have an impact on the requirements?	23	54	35	-	0.000	R4(–), R5(–), R7(–)
C1	Was the PM involved in making initial cost and effort estimates?	35	27	32	**	**	R4(-)
C2	Was the delivery decision made with appropriate requirements information?	73	17	51	+	0.000	R2, R4, R5, R7, R8, R10(–), C1
C3	Did the project have a schedule?	83	79	81	**	**	R2, R5
C4	If yes, were the effort and schedule estimates good?	49	12	33	+	0.000	R1, R2, R4, R5, R7, R8, R10(–), C2, C5(–), C6, C7(–)
C5	At some stage, were the developers involved in making estimates?	20	44	29	-	0.007	R2(–), R4(–), R7(–), R8(–), C2(–)
C6	Did the project have adequate staff to meet the schedule?	82	44	67	+	0.000	R1, R2, R4, R5, R7, R8, R10(–), C2, C5(–)
C7	Were staff added late to meet an aggressive schedule?	19	54	32	-	0.000	R4(–), R7(–), R10, C2(–), C6(–)
A1	Were potential risks identified at the project's start?	72	49	62	+	0.029	M1, M3, M5, M10, M12, R1, R2, R4, R5, R7, R8, C2, C4, C5(–), C6, C7(–), A2, A3
A2	Were risks incorporated into the project plan?	63	40	53	**	**	M1, M5, M10, M11, M12, R7, R8, C1, C2, C4, C6, C7(–)
A3	Were the risks managed throughout the project?	60	20	43	+	0.000	M1, M3, M4, M5, M9, M10, M12, R2, R4, R5, R7, R8, C2, C4, C5(–), C6, C7(–), A1, P1
P1	Was a postmortem review held?	35	21	29	NA	NA	R4, A1, A2, A3, P2
P2	If there was a postmortem review, were its results made available to other groups?	43	14	32	NA	NA	A1

tor of project success was M4 (Did the PM have a clear vision of the project?). This variable predicted 86 percent of successes, 56 percent of failures, and 75 percent correctly overall.

Requirements elicitation and management

Good project management necessitates complete, consistent requirements.[6] Although gathering requirements with a specific methodology (R1) was significantly associated with

We found that if requirements were initially incomplete, completing them during the project was positively associated with success.

requirements being complete and accurate at the project's start (R2), it wasn't significantly associated with project success. However, in 54 percent of our projects, respondents didn't know what requirements methodology their project's systems analysts used. Again, this didn't surprise us, because most respondents had no interaction with the systems analysts.

Requirements-gathering methods. Among the 46 percent of respondents who knew about requirements gathering, four projects used prototyping and nine used JAD (joint application design) sessions with prototyping. Eleven of these 13 projects were successful. Interviews and focus groups were the remaining projects' main requirements-gathering methods. Eight projects used UML to document requirements, but only three of these were successful. Practitioners commented that there were "too many new things without a pilot" and "unfamiliar methods." However, the failed UML projects had other problems such as poor estimates and no risk management, so their failures weren't necessarily due to using UML.

Managing changing requirements. Nearly half the projects began with incomplete requirements (R2); predictably, the scope changed for many of these projects during development (R6). A χ^2 test of R2 with R6 was significant. The scope was more likely to change for larger projects. Given that an organization needs control over the requirements function to advance from the lowest CMMI (Capability Maturity Model Integration, see www.sei.cmu.edu/cmmi) level, it's obvious that many of our sample's organizations are still at that level. These results agree with a survey Colin J. Neill and Philip A. Laplante conducted in 2002,[7] whose respondents thought that their companies didn't do enough requirements engineering. The number of projects that began with poor requirements suggests that these organizations should develop their projects using methodologies designed to deal with unclear requirements. However, this isn't the case.

The importance of requirements management. Our results, shown in Table 1, demonstrate that requirements continue to be an enormous problem for IS development and one of the most common causes of runaway projects.[8] Consistent with Robert Glass's observations,[9] we found that good requirements (R4) that were complete and accurate at the project's start (R2) with a well-defined project scope (R5), resulting in well-defined software deliverables (R9), were positively associated with project success. The importance of user involvement in requirements gathering (R7) supports observations by Glass[9] and Carl Clavedetschers.[10]

Although Barry Boehm includes a "continuing stream of requirements changes" in his top 10 risk items,[11] we didn't find an association between changing the scope during the project (R6) and project success. Instead, we found that if requirements were initially incomplete, completing them during the project (R3) was positively associated with success, as was being able to manage requirements and any changes to them through a central repository (R8). Only 60 percent of our projects used a central repository, supporting Clavedetscher's suggestion that software developers "fail to use requirements management to surface (early) errors or problems."[12]

When a project's size impacted requirements gathering (R10), project success was negatively influenced. This result agrees with another of Glass's observations,[9] suggesting that large projects hamper requirements gathering and lead to unclear, incomplete, and potentially unstable requirements.

Using logistic regression, we found that the best risk management predictor of project success was R4 (the requirements were good), which predicted 89 percent of successes, 58 percent of failures, and 78 percent correctly overall.

Cost and effort estimation and scheduling

Good cost and schedule estimates (C4) affect project success.[5] As early as 1975, Frederick Brooks stated that more projects have gone awry for lack of calendar time than from all other causes combined.[12] Optimistic estimation is still one of the two most common causes for runaway projects,[8] with cost and schedule failures exceeding any other kind of software failures in practice.[13] Boehm includes unrealistic schedules and budgets in his top 10 risk items.[11] However, having a schedule (C3) wasn't associated with project success; not having a schedule means you don't have to meet one.

Who's making the estimates? We initially assumed that the PM would be involved in de-

ciding the delivery date because he or she would likely know the project's technology, participants, and development practices better than anyone else. This assumption was mainly incorrect. Higher-level management, marketing, or the customer or user generally made initial estimates. The PM was involved in making initial cost and effort estimates for only 33 percent of the projects (C1), but responses to this question weren't associated with project success. PMs were able to negotiate the schedule in just under half (48 percent) of the projects where they weren't included in the initial delivery date or budget decisions.

In these organizations, the wrong people are making the estimates. This observation agrees with Glass.[9] When the PM had no say in project estimates, our respondents indicated that only 31 percent of those estimates were good. Given the lack of PM participation in estimation, it's understandable that only about half of the projects' delivery dates were made with appropriate requirements information (C2). Our result agrees with Glass[8] that most software estimates are performed at the beginning of the life cycle before the requirements phase and thus before the problem is understood. It's noteworthy that 76 percent of the estimates considered to be above average had some kind of PM input.

Estimate accuracy. Seventy-four percent of the projects were underestimated, 36 percent were accurately estimated, and no projects were overestimated. Overall, respondents thought that 38 percent of the estimates were either poor or very poor (33 percent of these projects were successful), 27 percent were average (66 percent successful), and 35 percent were either good or very good (86 percent successful). Many estimates initially thought to be of average quality were underestimates. Developers were more likely to contribute to project estimates when the requirements were incomplete, and developer involvement in making the estimates (C5) was negatively associated with project success. Furthermore, developers might not have a global perspective of the project, which could handicap them in producing projectwide estimates.

Project size was significantly related to whether a project had a schedule. Larger projects were more likely to have a schedule but were also more likely to have inadequate staffing levels, staff added late, and estimates that didn't take staff leave into account. Finally, adding staff late to meet an aggressive schedule (C7) was negatively related to project success, in agreement with Brooks's findings in *The Mythical Man Month*.[12]

Using logistic regression, we found that C2 (making delivery decisions with the appropriate requirements information) was the best C predictor of project outcomes, predicting 73 percent of successes, 83 percent of failures, and 77 percent correctly overall.

Risk assessment and postmortem reviews

Unfortunately, most developers and PMs perceive risk management processes and activities as creating extra work and expense, and risk management is the least-practiced project management discipline.[14] Respondents indicated that 33 percent of the projects had no risks, even though 62 percent of these projects failed. As Tom DeMarco and Timothy Lister noted, "if a project has no risks, don't do it."[5] Large projects were significantly less likely to have risks incorporated into their project plans. Active risk management characterizes software project management quality,[6] and managing risks throughout the project (A3) was significantly associated with project success.[5] The association between responses to questions A1, A2, and A3 and the PM being above average (M1) supports this observation.

Using logistic regression, we found that A3 (risks were managed throughout the project) predicted project success the best, predicting 60 percent of successes, 80 percent of failures, and 69 percent correctly overall.

Postmortem reviews are important for process improvement,[8] but companies seldom perform them. As a result, they tend to repeat the same mistakes project after project.[1,15] Few organizations conducted project postmortems, and those that conducted them didn't do so consistently (for our first 42 projects from the same organization, only 33 percent had postmortem reviews). Holding postmortem reviews (P1) was significantly associated with good requirements (R4) and managing risks throughout the process (A1, A2, and A3).

Discussion and recommendations

Our respondents' organizations rely heavily on software for many business functions. While we wouldn't assume our results are typ-

Postmortem reviews are important for process improvement, but companies seldom perform them. As a result, they tend to repeat the same mistakes.

The opportunity for greatest improvement is at a project's start, in the requirements and risk identification and control areas.

ical of all organizations, we believe they're reasonably typical of organizations that develop software in house. Surveys are based on self-reported data, which reflects what people say happened, not what they actually did or experienced. Because we surveyed software developers, our results are limited to their knowledge, attitudes, and beliefs regarding the projects and PMs with which they were involved. However, because most of the projects were fairly small, we believe that our respondents had a reasonable knowledge of most project events. The overall preponderance of small projects might, however, bias our results.

The best predictors of successful project outcomes, using logistic regression, were M4 (Did the project manager have a clear vision of the project?) with R4 (Overall, were the requirements good?) and C2 (Was the delivery decision made with adequate requirements information?). Combining these factors, 90 percent of successes, 70 percent of failures, and 82 percent of projects overall were predicted correctly.

We were surprised that so many projects started (and continued) with unclear requirements. Why are PMs prepared to go ahead with projects that don't have appropriate requirements, or without at least using a lifecycle methodology that can deal with unclear requirements? It's common wisdom that good requirements lead to software development success, so why are PMs prepared to jeopardize project success in this fashion? Poor requirements negatively affect the estimation process, leading to schedule and cost underestimates and inadequate staffing; staffing itself then becomes a major risk factor.

Business management seems to lack an appreciation of the steps necessary to successfully execute a project. Not only were PMs excluded from initial discussions, they subsequently weren't permitted to negotiate what the business managers had decided. Consequently, PMs are frequently short-circuited in managing their projects; they should be proactive in ensuring their participation in the estimation process rather than leaving it to someone else. Senior management needs better education regarding the importance of adequate requirements, systematic effort and schedule estimates, and the need to consult the PM for projects with fixed budgets. At the other end of the spectrum, developer input to the estimates improved neither the chances of success nor the estimates, probably because developers appeared to be asked for estimates only if the requirements were unclear.

A mismatch also exists between risk identification and control. Although most PMs identified risks at the project's start, fewer than half followed through during development. No respondent addressed risk assessment and management or the lack of it during our early discussions. Again, this underscores that risk management isn't routinely part of development.[5]

We found that managers viewed each project as a standalone entity and therefore didn't perceive postmortem reviews as important. Neither business managers nor project managers appeared to understand the specific causes of failed projects; consequently, they're unlikely to improve their performance on subsequent projects. Do these organizations have a culture of not admitting to mistakes? Managers ought to view finding out what went wrong with a project as a good thing, not something to hide. If PMs can't admit to or investigate failure, their projects will likely continue to fail.

Analysis of our survey suggests that a number of questions require further research. For example, what kinds of pressures lead PMs to not only start projects with poor requirements but also complete them without really knowing what the requirements are? How can PMs arrange to be part of estimation and scheduling, and how can they ensure that estimates are done with adequate requirements information? Why are risk planning and management so frequently ignored? And finally, why do so few organizations conduct postmortem reviews, and why don't they perform them consistently?

Table 1 shows that current practices are fair at best. The opportunity for greatest improvement is at a project's start, in the requirements and risk identification and control areas. We must face these issues if we wish to increase software project management quality and success.

References

1. R. Glass, "Project Retrospectives, and Why They Never Happen," *IEEE Software*, vol. 19, no. 5, 2002, pp. 112, 111.
2. J.M. Verner and W.M. Evanco, "An Investigation into Software Process Knowledge," *Managing Software En-*

gineering Knowledge, A. Aurum et al., eds., Springer-Verlag, 2003, pp. 29-47.
3. J. Jurison, "Software Project Management: The Manager's View," *Comm. Assoc. for Information Systems*, vol. 2, article 17, 1999; http://cais.isworld.org/articles/2-17/default.asp?View=html&x=33&y=6.
4. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
5. T. DeMarco and T. Lister, *Waltzing with Bears*, Dorset House, 2003.
6. J.S. Osmundson et al., "Quality Management Metrics for Software Development," *Information and Management*, vol. 40, no. 8, 2003, pp. 799–812.
7. C.J. Neill and P.A. Laplante, "Requirements Engineering: State of the Practice," *IEEE Software*, vol. 20, no. 6, 2003, pp. 40–45.
8. R. Glass, "Frequently Forgotten Fundamental Facts about Software Engineering," *IEEE Software*, vol. 18, no. 3, pp. 112, 110–111.
9. R. Glass, "How Not to Prepare for A Consulting Assignment and Other Ugly Consultancy Truths," *Comm. ACM*, vol. 41, no. 12, 1998, pp. 11–13.
10. C. Clavedetscher, "User Involvement Key to Success," *IEEE Software*, vol. 15, no. 2, 1998, pp. 30, 32.
11. B.W. Boehm, "Software Risk Management Principles and Practice," *IEEE Software*, vol. 8, no. 1, 1991, pp. 32–41.
12. F.P. Brooks Jr., *The Mythical Man Month: Essays on Software Engineering*, Addison-Wesley, 1975.
13. R. Glass, "Error-Free Software Remains Extremely Elusive," *IEEE Software*, vol. 20, no. 1, 2003, pp. 104, 103.
14. Y.H. Kwak and C.W. Ibbs, "Calculating Project Management's Return on Investment," *Project Management J.*, vol. 31, no. 2, 2000, pp. 38–47.
15. B. Collier, T. DeMarco, and P. Fearey, "A Defined Process for Project Postmortem Review," *IEEE Software*, vol. 13, no. 4, 1996, pp. 65–72.

About the Authors

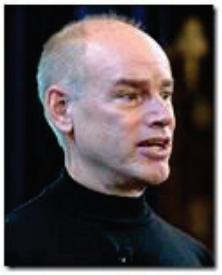
June M. Verner is a senior principal research scientist in the Empirical Software Engineering group at National Information and Communications Technology Ltd. Australia. Her research interests include software project management, risk management, and software process and product measurement. She received her PhD in software engineering from Massey University. She's a member of the IEEE Computer Society and the British Computer Society. Contact her at National ICT Australia, Australian Technology Park, Alexandria, Sydney, Australia; june.verner@nicta.com.au.

William M. Evanco is an associate professor in Drexel University's College of Information Science and Technology. His research interests include software product and process measurement and software performance analysis. He's also worked at Mitretek and MITRE in software development and intelligent transportation systems. He received his PhD in theoretical physics from Cornell University. Contact him at the College of Information Science and Technology, Drexel Univ., 3141 Chestnut St., Philadelphia, PA 19104; william.evanco@cis.drexel.edu.

De Scrumgids™
Ken Schwaber en Jeff Sutherland, 1991-2014

De Scrumgids™

De definitieve gids voor Scrum:
de regels van het spel



juli 2013

Ontwikkeld & onderhouden door Ken Schwaber en Jeff Sutherland

Scrum.org

Inhoudsopgave

Doel van de Scrumgids

Scrum is een raamwerk voor het ontwikkelen en onderhouden van complexe producten. Deze gids beschrijft de definitie van Scrum. Deze definitie bestaat uit de Scrum rollen, gebeurtenissen, artefacten en de regels die deze zaken samenbrengen. Ken Schwaber en Jeff Sutherland hebben Scrum ontwikkeld; de Scrumgids is door hen geschreven en ter beschikking gesteld. Zij staan samen achter deze gids.

Scrum Overzicht

Scrum (n): Een raamwerk waarbinnen mensen complexe, adaptieve problemen adresseren en tegelijkertijd op een productieve en creatieve wijze producten van de hoogst mogelijk waarde leveren.

Scrum is:

- Lichtgewicht
- Simpel om te begrijpen
- Extreem moeilijk om te leren beheersen

Scrum is een procesraamwerk dat sinds de jaren 1990 gebruikt wordt om complexe productontwikkeling te managen. Scrum is geen proces of techniek voor het bouwen van producten; het is een raamwerk waar binnen je de verschillende processen en technieken kunt inzetten. Scrum maakt de effectiviteit van jouw productmanagement en ontwikkeltechnieken inzichtelijk zodat je kunt verbeteren.

Het Scrum raamwerk bestaat uit Scrum Teams en hun bijbehorende rollen, gebeurtenissen, artefacten en regels. Elk onderdeel binnen het raamwerk dient een specifiek doel en is essentieel voor het gebruik en succes van Scrum.

De regels van Scrum verbinden de gebeurtenissen, rollen en artefacten met betrekking tot de interactie tussen hen. De regels van Scrum worden beschreven door deze hele tekst heen.

Specifieke strategieën voor het gebruik van het Scrum raamwerk verschillen en worden niet in deze gids beschreven.

Scrum Theorie

Scrum is gebaseerd op de theorie van empirische procesbesturing , ofwel het empirisme. Empirisme gaat er vanuit dat kennis ontstaat uit ervaring *en* het nemen van beslissingen op basis van wat bekend is. Scrum gebruikt een iteratieve, incrementele aanpak om voorspelbaarheid te optimaliseren en risico's te beheersen.

Drie pijlers vormen het fundament van elke implementatie van empirische procesbesturing: transparantie, inspectie en aanpassing.

Transparantie

Significante aspecten van het proces moeten zichtbaar zijn voor diegenen die verantwoordelijk zijn voor het resultaat. Transparantie vereist dat deze aspecten gedefinieerd zijn volgens een gezamenlijke standaard zodat waarnemers een gezamenlijk begrip hebben van wat er gezien wordt. Bijvoorbeeld:

- Een gemeenschappelijke taal met betrekking tot het proces moet door alle deelnemers worden gedeeld; en,
- Een gemeenschappelijke definitie van “Klaar” (“Definition of Done”)[1] moet worden gedeeld door degenen die het werk uitvoeren en degenen die het werkende product accepteren.

Inspectie

Scrum gebruikers moeten frequent de Scrum artefacten en de voortgang ten opzichte van het doel inspecteren, om ongewenste variantie te kunnen detecteren. Hun inspecties mogen niet zo frequent zijn dat de inspecties in de weg gaan zitten van het werk. Inspecties zijn het meest nuttig wanneer ze zorgvuldig worden uitgevoerd door vaardige inspecteurs, daar waar het werk gedaan wordt.

Aanpassing

Als een inspecteur bepaalt dat één of meer aspecten van een proces buiten de acceptabele limieten vallen en dat het resulterende product onacceptabel zal zijn, zal het proces of het onderhanden werk aangepast moeten worden. Een aanpassing moet zo snel mogelijk uitgevoerd worden om verdere afwijkingen te minimaliseren.

Scrum schrijft vier formele gelegenheden voor ten behoeve van inspectie en aanpassing, zoals beschreven in het *Scrum Gebeurtenissen* gedeelte van dit document.

- Sprint Planning
- Dagelijkse Scrum
- Sprint Review
- Sprint Retrospective

[1] Zie “Definitie van Klaar”, p. 16.

Het Scrum Team

Het Scrum Team bestaat uit een Product Owner, het Ontwikkelteam en een Scrum Master. Scrum Teams zijn zelforganiserend en multidisciplinair. Zelforganiserende teams kiezen zelf hoe zij het beste hun werk kunnen uitvoeren, in plaats van dat dit verteld wordt door iemand van buiten het team. Multidisciplinaire teams hebben alle competenties die nodig zijn om het werk uit te voeren, zonder afhankelijk te zijn van anderen buiten het team. Het team model in Scrum is ontworpen voor optimale flexibiliteit, creativiteit en productiviteit.

Scrum teams leveren iteratief en incrementeel producten, waarbij gelegenheden voor feedback gemaximaliseerd worden. Incrementele leveringen van een “Klaar” (Done) product zorgen ervoor dat een potentieel bruikbare versie van het product altijd beschikbaar is.

De Product Owner

De Product Owner is verantwoordelijk voor het maximaliseren van de waarde van het product en de werkzaamheden van het Ontwikkelteam. Hoe dit precies gedaan wordt verschilt enorm per organisatie, Scrum Team en individu.

De Product Owner is de enige persoon die verantwoordelijk is voor het managen van de Product Backlog. Product Backlog management omvat o.a.:

- Helder omschrijven van Product Backlog items;
- Ordenen van Product Backlog items, om doelen en missie op de beste manier te behalen;
- Optimaliseren van de waarde van het werk dat het Ontwikkelteam uitvoert;
- Ervoor zorgen dat de Product Backlog zichtbaar, transparant en duidelijk is voor iedereen, en dat het laat zien waar het Scrum Team als volgende aan gaat werken; en
- Ervoor zorgen dat het Ontwikkelteam de Product Backlog items begrijpt tot het niveau dat nodig is.

De Product Owner kan het bovenstaande werk zelf uitvoeren, of het door het Ontwikkelteam laten doen. In elk geval blijft de Product Owner verantwoordelijk.

De Product Owner is één persoon, geen comité. De Product Owner kan de wensen van een comité vertegenwoordigen via de Product Backlog, maar iedereen die een verandering wil in prioriteit van een Backlog Item, moet de Product Owner aanspreken.

Om te kunnen slagen als Product Owner, moet de gehele organisatie zijn of haar beslissingen respecteren. De beslissingen van de Product Owner zijn zichtbaar in de inhoud en ordening van de Product Backlog. Niemand mag het Ontwikkelteam aan een andere set van requirements laten werken het en het Ontwikkelteam is het niet toegestaan te acteren op basis van wat iemand anders zegt.

Het Ontwikkelteam

Het Ontwikkelteam bestaat uit professionals die het werk doen om een potentieel uitleverbaar Increment van het “Klaar” (Done) product op te leveren aan het einde van elke Sprint. Alleen leden van het Ontwikkelteam creëren het Increment.

Ontwikkelteams zijn zodanig gestructureerd en voorzien van bevoegdheden door de organisatie dat zij hun eigen werk kunnen organiseren en beheren. De resulterende synergie optimaliseert de algehele efficiëntie en effectiviteit van het team.

Ontwikkelteams hebben de volgende karakteristieken:

- Ze zijn zelfsturend. Niemand (zelfs de Scrum Master niet) vertelt het Ontwikkelteam hoe zij de Product Backlog moeten omzetten in Incrementen van potentieel uitleverbare functionaliteit;
- Ontwikkelteams zijn multidisciplinair, met alle benodigde vaardigheden om als team een product Increment te kunnen maken;
- Scrum erkent geen titels voor Ontwikkelteamleden anders dan Ontwikkelaar, ongeacht het werk dat door de persoon wordt uitgevoerd; er zijn geen uitzonderingen op deze regel;
- Ontwikkelteams omvatten geen subteams die toegewijd zijn aan een specifiek domein zoals testen of business analyse; er zijn geen uitzonderingen op deze regel; en,
- Individuele Ontwikkelteamleden kunnen specifieke vaardigheden of focusgebieden hebben, maar verantwoordelijkheid ligt bij het Ontwikkelteam als geheel.

Ontwikkelteam grootte

De optimale Ontwikkelteam grootte is klein genoeg om wendbaar te blijven en groot genoeg om significant werk te kunnen leveren binnen een Sprint. Minder dan drie Ontwikkelteamleden maakt dat interactie vermindert en resulteert in lagere productiviteitswinst. Kleinere Ontwikkelteams kunnen tegen een gebrek aan vaardigheden aanlopen tijdens de Sprint, wat resulteert in het niet kunnen opleveren van een potentieel uitleverbaar Increment. Meer dan negen leden in het team vereist teveel coördinatie. Grote Ontwikkelteams genereren teveel complexiteit om door een empirisch proces bestuurd te kunnen worden. De Product Owner en de Scrum Master worden hierin niet meegeteld tenzij zij ook werk van de Sprint Backlog uitvoeren.

De Scrum Master

De Scrum Master is ervoor verantwoordelijk dat Scrum wordt begrepen en goed wordt uitgevoerd. Scrum Masters doen dit door ervoor te zorgen dat het Scrum Team zich houdt aan de Scrum theorie, praktijk en regels.

De Scrum Master is een dienend leider voor het Scrum Team. De Scrum Master helpt diegenen buiten het Scrum Team te begrijpen welke van hun interacties met het Scrum Team behulpzaam zijn en welke niet. De Scrum Master helpt iedereen deze interacties te veranderen om zo de waarde die door het Scrum Team wordt gecreëerd te maximaliseren.

Scrum Master diensten aan de Product Owner

De Scrum Master dient de Product Owner op een aantal manieren, waaronder:

- Het vinden van technieken voor een effectief Product Backlog management;
- Het Scrum Team de noodzaak laten inzien om duidelijke en beknopte Product Backlog items te maken;
- Inzicht verkrijgen in de product planning in een empirische omgeving;
- Ervoor zorgdragen dat de Product Owner weet hoe de Product Backlog te ordenen zodat de maximale waarde verkregen kan worden;
- Inzicht verkrijgen in en het beoefenen van agility; en,
- Het faciliteren van Scrum gebeurtenissen wanneer gevraagd of nodig.

Scrum Master diensten aan het Ontwikkelteam

De Scrum Master dient het Ontwikkelteam op een aantal manieren, waaronder:

- Coachen van het Ontwikkelteam op het vlak van zelforganisatie en multidisciplinair werken;
- Het Ontwikkelteam helpen bij het maken van producten van hoge waarde;
- Het verwijderen van belemmeringen (‘impediments’) in de voortgang van het Ontwikkelteam;
- Het faciliteren van Scrum gebeurtenissen wanneer gevraagd of nodig; en,
- Het coachen van het Ontwikkelteam in organisatorische omgevingen waarbinnen Scrum nog niet volledig is opgenomen en begrepen.

Scrum Master diensten aan de Organisatie

De Scrum Master dient de organisatie op een aantal manieren, waaronder:

- Het leiden en coachen van de organisatie in haar Scrum adoptie;
- Plannen van Scrum implementaties in de organisatie;
- Helpen van medewerkers en belanghebbenden bij het begrijpen en doorleven van Scrum en empirische productontwikkeling;
- Initiëren van veranderingen die de productiviteit van het Scrum Team verhogen; en,
- Met andere Scrum Masters samenwerken om de effectiviteit van de toepassing van Scrum binnen de organisatie te verhogen.

Scrum Gebeurtenissen

Binnen Scrum worden voorgeschreven gebeurtenissen gebruikt om regelmaat te creëren en om de behoefte aan overige, niet in Scrum gedefinieerde bijeenkomsten te minimaliseren. Scrum maakt gebruik van timeboxes bij gebeurtenissen, zodat elke gebeurtenis aan een maximale tijdsduur is gebonden. Als een Sprint eenmaal begonnen is, is de duur niet meer aanpasbaar en kan dus niet ingekort of verlengd worden. De andere gebeurtenissen mogen eindigen zodra het

doel van de gebeurtenis is bereikt, zodat de juiste tijd wordt gespendeerd aan planning zonder waardevermindering (“waste”) van het planning proces.

Anders dan de Sprint zelf, die een container is voor alle andere gebeurtenissen, is elke gebeurtenis in Scrum een formele gelegenheid om iets te inspecteren en aan te passen. Deze gebeurtenissen zijn specifiek ontworpen om kritieke transparantie en inspectie mogelijk te maken. Het niet opnemen van één van de deze gebeurtenissen resulteert in een vermindering van transparantie en is een gemiste kans voor inspectie en aanpassing.

De Sprint

Het hart van Scrum is een Sprint, een timebox van één maand of minder waarbinnen een “Klaar”, bruikbaar en potentieel uitleverbaar product Increment wordt gecreëerd. Sprints hebben idealiter een consistente lengte gedurende de ontwikkelinspanning. Een nieuwe Sprint start direct nadat de vorige Sprint is afgesloten.

Sprints bevatten en bestaan uit een Sprint Planningsbijeenkomst, Dagelijkse Scrums, het ontwikkelwerk, de Sprint Review en de Sprint Retrospective.

Gedurende de Sprint:

- Worden geen veranderingen aangebracht die het Sprint Doel in gevaar kunnen brengen;
- Verminderen kwaliteitsdoelstellingen niet; en,
- Mag de scope worden verduidelijkt en heronderhandeld tussen Product Owner en Ontwikkelteam naarmate meer is geleerd.

Elke Sprint mag worden beschouwd als een project met een horizon van niet meer dan één maand. Net als projecten worden Sprints gebruikt om iets te bereiken. Elke Sprint bestaat uit een definitie van wat er gemaakt moet worden, een ontwerp en een flexibel plan dat de bouw richting geeft, de werkzaamheden zelf en het resulterende product.

Sprints worden beperkt tot één kalendermaand. Wanneer een Sprinthorizon te ver weg is kan de definitie van wat gemaakt wordt veranderen, complexiteit kan verhogen en risico’s kunnen vergroot worden. Sprints geven de mogelijkheid van voorspelbaarheid door, op zijn minst elke maand, ervoor te zorgen dat via inspectie en aanpassing naar een Sprint Doel wordt gestuurd. Sprints zorgen er ook voor dat het risico wordt beperkt tot maximaal één maand aan kosten.

Afbreken van een Sprint

Een Sprint kan worden afgebroken voordat de Sprint timebox voorbij is. Alleen de Product Owner heeft de autoriteit om een Sprint af te breken, hoewel hij of zij dit kan doen onder invloed van de belanghebbenden, het Ontwikkelteam of de Scrum Master.

Een Sprint wordt typisch afgebroken als het Sprint Doel achterhaald is geworden. Dit kan gebeuren als de organisatie van richting verandert of indien de markt- of technologieomstandigheden veranderen. In zijn algemeenheid zou een Sprint moeten worden

afgebroken indien deze, gegeven de omstandigheden, niet zinvol meer is. Echter, gezien de korte looptijd van een Sprint, is het afbreken zeer zelden zinvol.

Indien een Sprint wordt afgebroken, worden Product Backlog items die “Klaar” zijn geïnspecteerd. Indien een deel van het werk potentieel uitleverbaar is zal de Product Owner dit normaal gesproken accepteren. Alle incomplete Product Backlog Items worden opnieuw ingeschat en terug op de Product Backlog gezet. Het werk dat reeds gedaan is voor deze items vermindert snel in waarde en moet frequent opnieuw worden ingeschat.

Sprints afbreken kost resources, want iedereen moet hergroeperen in een nieuwe Sprint Planning om een nieuwe Sprint te starten. Het Afbreken van een Sprint is vaak traumatisch voor het Scrum Team en komt zelden voor.

Sprint Planning

Het werk dat uitgevoerd moet worden tijdens een Sprint wordt gepland tijdens de Sprint Planning. Het maken van dit plan is een gezamenlijke inspanning van het gehele Scrum Team.

De Sprint Planningsbijeenkomst is een meeting binnen een timebox van acht uur voor een Sprint van één maand. Voor kortere Sprints is de bijeenkomst normaliter korter. De Scrum Master draagt er zorg voor dat deze gebeurtenis plaats vindt en dat de deelnemers het doel begrijpen. De Scrum Master leert het Scrum Team hoe ze het binnen de timebox kunnen afronden.

Sprint Planning beantwoordt de volgende vragen:

- Wat kan worden geleverd in het resulterende Increment van de komende Sprint?
- Hoe wordt het benodigde werk om dit Increment te leveren behaald?

Onderwerp één: Wat kan worden gedaan deze Sprint?

In dit deel werkt het Ontwikkelteam aan een voorspelling van de functionaliteit die in deze Sprint ontwikkeld zal worden. De Product Owner bespreekt het doel dat bereikt zou moeten worden met de Sprint en de Product Backlog items die, als ze worden afgemaakt tijdens de Sprint, het Sprint Doel zullen vervullen. Het hele Scrum Team werkt samen om het werk van de Sprint te begrijpen.

De input voor deze bijeenkomst is de Product Backlog, het laatste product Increment, de verwachte capaciteit van het Ontwikkelteam gedurende de Sprint en de prestaties uit het verleden van het Ontwikkelteam. Het aantal items dat wordt geselecteerd van de Product Backlog is volledig aan het Ontwikkelteam. Alleen het Ontwikkelteam kan inschatten wat zij kan bereiken binnen de aankomende Sprint.

Nadat het Ontwikkelteam een prognose heeft gegeven van de Product Backlog items die zij kunnen gaan leveren, formuleert het Scrum Team een Sprint Doel. Het Sprint Doel is een doel dat binnen de sprint wordt behaald door het implementeren van de Product Backlog, en het geeft sturing aan het Team over het waarom het huidige increment gebouwd wordt.

Onderwerp twee: Hoe wordt het gekozen werk gedaan?

Na het Sprint Doel te hebben vastgesteld en het werk voor de Sprint te hebben geselecteerd, beslist het Ontwikkelteam hoe het deze functionaliteit realiseert tot een “Klaar” product Increment gedurende de Sprint. De voor de Sprint geselecteerde Product Backlog items plus het plan voor het leveren ervan wordt de Sprint Backlog genoemd.

Gebruikelijk is dat het Ontwikkelteam start met het ontwerpen van het systeem en het werk dat gedaan moet worden om de Product Backlog om te zetten in een werkend product Increment. Werk kan variëren in grootte of geschatte inspanning. Tijdens de Sprint Planningsbijeenkomst plant het Ontwikkelteam zodanig veel werk dat een voorspelling gedaan kan worden met betrekking tot wat het denkt te kunnen opleveren in de komende Sprint. Aan het einde van deze meeting heeft het team het werk voor de eerste dagen van de Sprint opgedeeld in eenheden, vaak van één dag of kleiner. Het Ontwikkelteam organiseert zichzelf om het werk in de Sprint Backlog gedaan te krijgen, zowel gedurende de Sprint Planning als gedurende de Sprint.

De Product Owner kan helpen om de geselecteerde Product Backlog items verder te verduidelijken en om afwegingen te maken. Indien het Ontwikkelteam bepaalt dat er te veel of te weinig werk is, mag het team over de Sprint Backlog items opnieuw onderhandelen met de Product Owner. Het Ontwikkelteam mag ook andere mensen uitnodigen om advies te leveren op technisch vlak of over het domein.

Aan het einde van de Sprint Planningsbijeenkomst zou het Ontwikkelteam in staat moeten zijn om aan de Product Owner en Scrum Master uit te leggen hoe zij van plan zijn, als zelf organiserend team, het Sprintdoel te behalen en het verwachte Increment te realiseren.

Sprint Doel

Het Sprint Doel is een doelstelling welke gesteld wordt voor de Sprint, welke kan worden behaald door het implementeren van de Product Backlog. Het geeft richting aan het Ontwikkelteam op het “waarom” het dit increment aan het bouwen is. Het wordt gecreëerd gedurende de Sprint Planning. Het Sprint Doel geeft het Ontwikkelteam de nodige flexibiliteit ten aanzien van de functionaliteit die geïmplementeerd wordt in de Sprint. De geselecteerde Product Backlog items leveren een samenhangende functie, wat het Sprint Doel kan zijn. Het Sprint Doel kan elke andere samenhang zijn die ervoor zorgt dat het Ontwikkelteam gaat samenwerken in plaats van aan verschillende initiatieven.

Tijdens het werken houdt het Ontwikkelteam het Sprint Doel in het oog. Functionaliteit en technologie worden geïmplementeerd om het Sprint Doel te bereiken. Indien het werk anders blijkt te zijn dan het Ontwikkelteam had verwacht, werken zij samen met de Product Owner om over de scope van de Sprint Backlog binnen deze Sprint te onderhandelen.

Dagelijkse Scrum

De Dagelijkse Scrum is een 15-minuten-timeboxed gebeurtenis voor het Ontwikkelteam om activiteiten te synchroniseren en een plan te maken voor de komende 24 uur. Dit wordt gedaan

door het werk sinds de laatste Dagelijkse Scrum te inspecteren en te voorspellen welk werk gedaan kan worden tot de volgende Dagelijkse Scrum.

De Dagelijkse Scrum wordt elke dag gehouden op dezelfde tijd en plaats om complexiteit te reduceren. Gedurende de bijeenkomst licht elk Ontwikkelteamlid het volgende toe:

- Wat heb ik gisteren gedaan dat het Ontwikkelteam heeft geholpen het Sprint Doel te bereiken?
- Wat ga ik vandaag doen om het Ontwikkelteam te helpen het Sprint Doel te bereiken?
- Zie ik enig obstakel die mij of het Ontwikkelteam in de weg staat het Sprint Doel te bereiken?

Het Ontwikkelteam gebruikt de Dagelijkse Scrum om te voortgang ten opzichte van het Sprint Doel te beoordelen en om te beoordelen hoe de trend is van het gedane werk ten opzichte van de Sprint Backlog. De Dagelijkse Scrum optimaliseert de waarschijnlijkheid dat het Ontwikkelteam het Sprint Doel behaalt. Elke dag moet het Ontwikkelteam begrijpen hoe zij samen gaan werken als zelf organiserend team om het Sprint Doel te behalen en hoe het verwachte Increment te realiseren in het restant van de Sprint. Vaak houdt het Ontwikkelteam ,of teamleden een bijeenkomst direct na de Dagelijkse Scrum voor gedetailleerde discussies, of om de rest van het werk in de Sprint aan te passen of te herplannen.

De Scrum Master zorgt ervoor dat het Ontwikkelteam de bijeenkomst houdt, maar het Ontwikkelteam zelf is verantwoordelijk voor het uitvoeren van de Dagelijkse Scrum. De Scrum Master leert het Ontwikkelteam om de Dagelijkse Scrum binnen de 15-minuten-timebox te houden.

De Scrum Master dwingt de regel af dat alleen Ontwikkelteamleden participeren in de Dagelijkse Scrum.

Dagelijkse Scrums verbeteren communicatie, elimineren andere bijeenkomsten, identificeren obstakels bij de ontwikkeling zodat die verwijderd kunnen worden, belichten en bevorderen het maken van snelle beslissingen en verbeteren het kennisniveau van het Ontwikkelteam. Dit is een zeer belangrijke “inspect and adapt” (inspectie en aanpassing) bijeenkomst.

Sprint Review

Een Sprint Review wordt gehouden aan het einde van de Sprint om het Increment te inspecteren en indien nodig de Product Backlog aan te passen. Gedurende de Sprint Review bekijken het Scrum Team en de belanghebbenden samen wat er bereikt is gedurende de Sprint. Op basis hiervan en op basis van veranderingen in de Product Backlog, werken de aanwezigen samen aan de volgende stappen die genomen kunnen worden om waarde te optimaliseren. Dit is een informele bijeenkomst, geen status meeting, en de presentatie van het Increment heeft als doel feedback te verzamelen en samenwerking te bevorderen.

Dit is een vier-uur-timebox bijeenkomst voor Sprints van één maand. Over het algemeen wordt er minder tijd gepland voor kortere sprints. De Scrum Master draagt er zorg voor dat de gebeurtenis plaatsvindt en dat de aanwezigen het doel begrijpen. De Scrum Master leert iedereen om het binnen de timebox te houden.

De Sprint Review omvat de volgende elementen:

- De aanwezigen zijn het Scrum Team en belangrijke belanghebbenden die worden uitgenodigd door de Product Owner.
- De Product Owner identificeert welke Product Backlog items "Klaar" zijn en wat er niet "Klaar" is;
- Het Ontwikkelteam bespreekt wat er goed ging gedurende de Sprint, welke problemen ze zijn tegengekomen en hoe deze problemen werden opgelost;
- Het Ontwikkelteam demonstreert het werk dat "Klaar" is en beantwoordt vragen met betrekking tot het Increment;
- De Product Owner bespreekt de Product Backlog zoals deze nu is. Hij of zij projecteert waarschijnlijke data van completering op basis van de voortgang tot nu toe (als nodig);
- De gehele groep werkt samen aan wat er vervolgens gemaakt gaat worden, zodat de Sprint Review waardevolle input levert voor de komende Sprint Planningsbijeenkomst;
- Een beoordeling van de markt of potentieel gebruik van het product kan beïnvloeden wat het meest waardevolle is om vervolgens te maken; en,
- Een overzicht van de tijdlijn, budget, potentiele mogelijkheden, en markt voor de volgende verwachte ingebruikname van het product.

Het resultaat van de Sprint Review is een herziene Product Backlog welke de waarschijnlijke Product Backlog items voor de volgende Sprint definieert. De Product Backlog kan ook over het geheel worden aangepast om nieuwe kansen te kunnen omarmen.

Sprint Retrospective

De Sprint Retrospective is een kans voor het Scrum Team om zichzelf te inspecteren en een plan te maken om zichzelf gedurende de komende Sprint te verbeteren.

De Sprint Retrospective vindt plaats na de Sprint Review en vóór de volgende Sprint Planning. Dit is een drie-uur-timebox bijeenkomst voor Sprints van één maand. Over het algemeen wordt minder tijd gepland voor kortere sprints. De Scrum Master draagt er zorg voor dat de gebeurtenis plaatsvindt en dat de aanwezigen het doel begrijpen. De Scrum Master leert iedereen om het binnen de timebox te houden. De Scrum Master neemt deel als een gelijkwaardig teamlid aan de bijeenkomst vanuit zijn verantwoordelijkheid voor het Scrum proces.

Het doel van de Sprint Retrospective is om:

- Te inspecteren hoe de laatste Sprint is gegaan met betrekking tot mensen, relaties, processen en tools;
- Dingen die goed gingen en potentiële verbeteringen te identificeren en te ordenen; en,
- Een plan te creëren om verbeteringen op de manier waarop het Scrum Team zijn werk doet te implementeren.

De Scrum Master moedigt het Scrum Team aan om, binnen het Scrum proces raamwerk, hun ontwikkelproces en practices te verbeteren, om het team effectiever en plezieriger te maken voor de volgende Sprint. Gedurende elke Sprint Retrospective plant het Scrum Team manieren om de kwaliteit van het product te verhogen door zo nodig de definitie van “Klaar” aan te passen.

Tegen het einde van de Sprint Retrospective zou het Scrum Team verbeteringen moeten hebben benoemd die geïmplementeerd zullen worden in de volgende Sprint. Het implementeren van deze verbeteringen in de volgende Sprint is de aanpassing naar aanleiding van de inspectie van het Scrum Team zelf. Alhoewel verbeteringen altijd geïmplementeerd mogen worden, biedt de Sprint Retrospective een formeel moment gericht op inspectie en aanpassing.

Scrum Artefacten

De artefacten van Scrum vertegenwoordigen werk of waarde die waardevol zijn voor het bieden van transparantie en voor mogelijkheden tot inspectie en adaptie. De artefacten die Scrum definieert zijn specifiek ontworpen voor maximale transparantie van sleutelinformatie zodat iedereen hetzelfde begrip heeft van het artefact.

Product Backlog

De Product Backlog is een geordende lijst van alles dat mogelijk nodig is in het product, en is de enige bron van requirements voor wijzigingen die aan het product gemaakt moeten worden. De Product Owner is verantwoordelijk voor de Product Backlog, inclusief de inhoud, beschikbaarheid en ordening.

Een Product Backlog is nooit compleet. De eerste versies ervan bevatten alleen de initieel bekende en best begrepen requirements. De Product Backlog ontwikkelt zich naar gelang het product en de omgeving waarin het gebruikt gaat worden zich verder ontwikkelen. De Product Backlog is dynamisch: hij verandert voortdurend om duidelijk te maken wat het product nodig heeft om toepasbaar, concurrerend en bruikbaar te zijn. Zolang het product bestaat, bestaat de bijbehorende Product Backlog ook.

De Product Backlog somt alle kenmerken, functies, requirements, verbeteringen en foutherstel op die gezamenlijk de wijzigingen zijn die in toekomstige releases aan het product gemaakt moeten worden. Product Backlog items hebben als kenmerken een beschrijving, ordening, een schatting en een waarde.

Vanaf het moment dat het product gebruikt wordt en waarde oplevert, en de markt terugkoppeling levert, wordt de Product Backlog een grotere en meer uitputtende lijst. Requirements blijven altijd veranderen, en dus is de Product Backlog een levend artefact. Veranderingen in bedrijfseisen, marktomstandigheden of technologieën kunnen veranderingen in de Product Backlog tot gevolg hebben.

Meerdere Scrum Teams werken vaak samen aan hetzelfde product. Één Product Backlog wordt gebruikt om het aankomende werk op het product te beschrijven. Er wordt dan een Product Backlog attribuut gebruikt om items te groeperen.

Product Backlog verfijning (“refinement”) is het toevoegen van detail, schattingen en volgorde aan de items op de Product Backlog. Dit is een doorlopend proces waarbij de Product Owner en het Ontwikkelteam samenwerken aan de details van de Product Backlog items. Gedurende Product Backlog onderhoud worden items gereviewed en bijgewerkt. Refinement neemt gewoonlijk niet meer dan 10% van de capaciteit van het Ontwikkelteam in beslag. Echter, Product Backlog items kunnen op elk moment worden bijgewerkt door de Product Owner of naar de Product Owner’s eigen beoordeling.

Product Backlog items met een hogere rangorde zijn normaliter duidelijker en meer gedetailleerd dan die met een lagere rangorde. De schattingen zijn meer precies vanwege de hogere mate van duidelijkheid en detaillering. Hoe lager de rangorde, hoe minder details. De Product Backlog items waarmee het Ontwikkelteam de komende Sprint aan de slag gaat zijn zo ver verfijnd dat elk item “Klaar” kan zijn binnen een Sprint. De Product Backlog items die door het Ontwikkelteam “Klaar” kunnen zijn binnen een Sprint worden beschouwd als “Ready” voor selectie in een Sprint Planningsbijeenkomst. Product Backlog items verkrijgen deze mate van transparantie over het algemeen door de verfijningsactiviteiten die hierboven besc hreven staan.Het Ontwikkelteam is verantwoordelijk voor alle schattingen. De Product Owner mag het Ontwikkelteam beïnvloeden door te helpen bij het begrijpen en maken van afwegingen, maar de mensen die het werk moeten doen maken de uiteindelijke schatting.

Controleren op de voortgang tot een doel

Op elk moment in de tijd kan het totale werk, dat nog nodig is om een doel te bereiken, worden opgeteld. De Product Owner houdt de totale resterende hoeveelheid werk bij, op zijn minst voor elke Sprint Review. De Product Owner vergelijkt dit totaal met de resterende totale hoeveelheid werk uit eerdere Sprint Reviews om een inschatting te maken of het resterende werk binnen de gewenste tijd kan zijn afgerond voor het gestelde doel. Deze informatie wordt transparant gemaakt voor alle stakeholders.

Verschillende technieken voor trendanalyse: burndown, burnup en andere predictie-methoden zijn gebruikt om voortgang te voorspellen. Deze zijn nuttig gebleken. Echter, ze vervangen niet het belang van empirisme. In complexe omgevingen is niet bekend wat er in de toekomst gaat gebeuren. Alleen wat er is gebeurd mag gebruikt worden voor toekomstgerichte besluitvorming.

Sprint Backlog

De Sprint Backlog is de verzameling Product Backlog items geselecteerd voor de Sprint inclusief het plan voor opleveren van het product Increment en voor realisatie van het Sprint Doel. De Sprint Backlog is een voorspelling door het Ontwikkelteam over de functionaliteit die aanwezig zal zijn in het volgende Increment en het werk dat nodig is om die functionaliteit te leveren in een “Klaar” Increment.

De Sprint Backlog maakt al het werk zichtbaar dat het Ontwikkelteam heeft geïdentificeerd als noodzakelijk voor behalen van het Sprint Doel.

De Sprint Backlog is een plan met voldoende detail zodat veranderingen in de voortgang begrepen kunnen worden in de Dagelijkse Scrum. Het Ontwikkelteam past de Sprint Backlog gedurende de Sprint aan, en de Sprint Backlog ontwikkelt zich gedurende de Sprint. Deze ontwikkeling gebeurt als het Ontwikkelteam het plan afwerkt en meer leert over het werk dat nodig is om het Sprintdoel te halen.

Als er nieuw werk nodig is voegt het Ontwikkelteam dat toe aan de Sprint Backlog. Wanneer werk wordt uitgevoerd of afgerond wordt de schatting van het resterende werk bijgesteld. Als onderdelen van het plan overbodig blijken, worden ze verwijderd. Alleen het Ontwikkelteam kan haar Sprint Backlog bijwerken gedurende een Sprint. De Sprint Backlog is een zeer zichtbaar, real-time beeld van het werk dat het Ontwikkelteam van plan is te doen gedurende de Sprint, en het behoort uitsluitend toe aan het Ontwikkelteam.

Controleren Sprint voortgang

Op elk moment in de Sprint kan de totaal resterende hoeveelheid werk voor de Sprint Backlog items worden opgeteld. Het Ontwikkelteam houdt dit totaal minstens voor elke Dagelijkse Scrum bij om de waarschijnlijkheid van het halen van het Sprintdoel te projecteren. Door het bijhouden van de resterende hoeveelheid werk gedurende de Sprint kan het Ontwikkelteam haar voortgang bewaken.

Increment

Het Increment is het totaal van alle Product Backlog items voltooid tijdens een Sprint en alle voorgaande Sprints. Aan het eind van een Sprint moet het nieuwe Increment “Klaar” zijn, wat betekent dat het in bruikbare toestand is en voldoet aan de Definitie van “Klaar” gebruikt door het Scrum Team. Het moet in bruikbare conditie zijn ongeacht of de Product Owner daadwerkelijk besluit het te in gebruik te nemen.

Artefact Transparantie

Scrum is gebouwd op transparantie. Beslissingen die genomen worden om de waarde te optimaliseren en risico’s te beheersen, worden genomen op basis van de waargenomen toestand van de artefacten. Voor zover de transparantie compleet is, hebben deze beslissingen

een goede basis. Als de artefacts niet geheel transparant zijn, kunnen deze beslissingen verkeerd zijn, neemt waarde af en neemt risico toe.

De Scrum Master werkt met de Product Owner, het Ontwikkelteam, en andere betrokken partijen om duidelijk te krijgen of de artefacten volledig transparant zijn. Er zijn practices voor het omgaan met onvolledige transparantie; de Scrum Master moet iedereen helpen om de meest toepasselijke practices te gebruiken in geval van absentie van volledige transparantie. Een Scrum Master kan onvolledige transparantie detecteren door de artefacten te inspecteren, patronen te herkennen, nauwkeurig te luisteren en verschillen te vinden tussen verwachtingen en resultaten.

Het is de taak van de Scrum Master om met het Scrum Team en de organisatie de transparantie van de artefacten de vergroten. Dit werk behelst meestal leren, overtuigen en verandering. Transparantie zal niet plotsklaps plaatsvinden; het is een traject.

Definitie van “Klaar” (Definition of “Done”)

Indien een Product Backlog item wordt omschreven als “Klaar”, moet iedereen begrijpen wat “Klaar” betekent. Hoewel dit significant verschilt per Scrum Team, moeten de teamleden een gezamenlijk begrip hebben wat het betekent om het werk klaar te hebben, om transparantie te kunnen garanderen. Deze “Definitie van klaar” (“Definition of Done”) voor het Scrum Team wordt gebruikt om te controleren wanneer het werk voor een product Increment klaar is.

Dezelfde definitie helpt het Ontwikkelteam te bepalen hoeveel Product Backlog items zij kunnen selecteren tijdens de Sprint Planning. Het doel van elke Sprint is om Incrementen van potentieel opleverbare functionaliteit te leveren die voldoen aan de huidige Definitie van “Klaar” van het Scrum Team.

Ontwikkelteams leveren elke Sprint een Increment van product functionaliteit . Dit Increment is bruikbaar, zodat een Product Owner kan besluiten dit onmiddellijk in gebruik te nemen. Als de ontwikkelorganisatie al een Definitie van “Klaar” **heeft** als conventie, standaard of richtlijn, zullen alle Scrum Teams deze tenminste moeten volgen. Als de ontwikkelorganisatie nog **geen** Definitie van “Klaar” **heeft** als conventie, standaard of richtlijn, zal het Ontwikkelteam van het Scrum Team zelf een Definitie van “Klaar” moeten definiëren die geschikt is voor het product. Als er meerdere Scrum Teams werken aan hetzelfde systeem of productrelease, moeten de Ontwikkelteams van alle Scrum Teams gezamenlijk de Definitie van “Klaar” definiëren.

Elk Increment is additief aan alle voorgaande Incrementen en grondig getest zodat wordt gegarandeerd dat alle Incrementen samen werken.

Naarmate Scrum Teams meer volwassen worden, is de verwachting dat hun Definitie van “Klaar” uitgebreid wordt met striktere criteria ten behoeve van een hogere kwaliteit. Elk product of systeem zou een Definitie van “Klaar” moeten hebben die als standaard geldt voor elk werk wat eraan gedaan wordt.

Eindnoot

Scrum is gratis en wordt aangeboden middels deze gids. De rollen, artefacten, gebeurtenissen en regels van Scrum staan vast. Hoewel het implementeren van delen van Scrum mogelijk is, is het daaruit volgend resultaat geen Scrum. Scrum bestaat slechts als geheel en functioneert goed als container voor andere technieken, methodologieën en practices.

Erkenning

Mensen

Van de duizenden mensen die hebben bijgedragen aan Scrum, moeten we toch een paar mensen uitlichten die sleutelrollen hebben vervuld in de eerste tien jaar. Als eerste Jeff Sutherland , samenwerkend met Jeff McKenna, en Ken Schwaber samen met Mike Smith en Chris Martin. En dan natuurlijk nog vele anderen, zonder wiens hulp Scrum niet verfijnd zou zijn tot wat het vandaag is.

Geschiedenis

Ken Schwaber en Jeff Sutherland hebben samen Scrum voor het eerst gepresenteerd tijdens de OOPSLA conferentie in 1995. Deze presentatie documenteerde datgene wat Ken en Jeff geleerd hadden gedurende de voorgaande jaren bij het toepassen van Scrum.

De geschiedenis van Scrum wordt nu al beschouwd als lang. Ter ere van de eerste plekken waar het werd geprobeerd en verfijnd noemen we hier Individual, Inc., Fidelity Investments, en IDX (nu GE Medical).

De Scrumgids documenteert Scrum zoals ontwikkeld en ruim twintig jaar in stand gehouden door Jeff Sutherland en Ken Schwaber. Andere bronnen voorzien u van patterns, processen en inzichten die het Scrum raamwerk ondersteunen. Deze optimaliseren de productiviteit, waarde, creativiteit, en trots

Vertaling

Deze gids is vertaald vanuit de originele Engelse versie, ter beschikking gesteld door Ken Schwaber en Jeff Sutherland. Deze vertaling is gemaakt door Stefan Warringa , Eelco Gravendeel en Ruud Rietveld.

Artikel 8 bij leereenheid 3

Scrum Development Process
Ken Schwaber

Advanced Development Methods, Inc., 1995

Scrum Development Process

Ken Schwaber
Advanced Development Methods, Inc., 1995

Abstract

The stated, accepted philosophy for systems development is that the development process is a well understood approach that can be planned, estimated, and successfully completed. This has proven incorrect in practice. SCRUM assumes that the systems development process is an unpredictable, complicated process that can only be roughly described as an overall progression. SCRUM defines the systems development process as a loose set of activities that combines known, workable tools and techniques with the best that a development team can devise to build systems. Since these activities are loose, controls to manage the process and inherent risk are used. SCRUM is an enhancement of the commonly used iterative/incremental object-oriented development cycle.

KEY WORDS: Scrum SEI Capability-Maturity-Model Process Empirical

---

1. Introduction

In this paper we introduce a development process, Scrum, that treats major portions of systems development as a controlled black box. We relate this to complexity theory to show why this approach increases flexibility and ability to deal with complexity, and produces a system that is responsive to both initial and additionally occurring requirements.

Numerous approaches to improving the systems development process have been tried. Each has been touted as providing "significant productivity improvements." All have failed to produce dramatic improvements [41]. As Grady Booch noted, "We often call this condition the software crisis, but frankly, a malady that has carried on this long must be called normal."[42]

Concepts from industrial process control are applied to the field of systems development in this paper. Industrial process control defines processes as either "theoretical" (fully defined) or "empirical" (black box). When a black box process is treated as a fully defined process, unpredictable results occur [43]. A further treatment of this is provided in Appendix 1.

A significant number of systems development processes are not completely defined, but are treated as though they are. Unpredictability without control results. The Scrum approach treats these systems development processes as a controlled black box.
Variants of the Scrum approach for new product development with high performance small teams was first observed by Takeuchi and Nonaka [1][3] at Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox, and Hewlett-Packard. A similar approach applied to software

development at Borland was observed by Coplien [2] to be the highest productivity C++ development project ever documented. More recently, a refined approach to the SCRUM process has been applied by Sutherland [44] to Smalltalk development and Schwaber [45] to Delphi development.

The Scrum approach is used at leading edge software companies with significant success. We believe Scrum may be appropriate for other software development organizations to realize the expected benefits from Object Oriented techniques and tools [46].

2. Overview

Our new approach to systems development is based on both defined and black box process management. We call the approach the Scrum methodology (see Takeuchi and Nonaka, 1986), after the Scrum in Rugby -- a tight formation of forwards who bind together in specific positions when a Scrumdown is called.

As will be discussed later, Scrum is an enhancement of the iterative and incremental approach to delivering object-oriented software initially documented by Pittman [47] and later expanded upon by Booch [48]. It may use the same roles for project staff as outlined by Graham [49], for example, but it organizes and manages the team process in a new way.

Scrum is a management, enhancement, and maintenance methodology for an existing system or production prototype. It assumes existing design and code which is virtually always the case in object-oriented development due to the presence of class libraries. Scrum will address totally new or re-engineered legacy systems development efforts at a later date.

Software product releases are planned based on the following variables:

- Customer requirements - how the current system needs enhancing.
- Time pressure - what time frame is required to gain a competitive advantage.
- Competition - what is the competition up to, and what is required to best them.
- Quality - what is the required quality, given the above variables.
- Vision - what changes are required at this stage to fulfill the system vision.
- Resource - what staff and funding are available.

These variables form the initial plan for a software enhancement project. However, these variables also change during the project. A successful development methodology must take these variables and their evolutionary nature into account.

3. Current Development Situation

Systems are developed in a highly complicated environment. The complexity is both within the development environment and the target environment. For example, when the air traffic control system development was initiated, three-tier client server systems and airline deregulation did not have to be considered. Yet, these environmental and technical changes occurred during the project and had to be taken into account within the system being built.

Environmental variables include:

- Availability of skilled professionals - the newer the technology, tools, methods, and domain, the smaller the pool of skilled professionals.
- Stability of implementation technology - the newer the technology, the lower the stability and the greater the need to balance the technology with other technologies and manual procedures.
- Stability and power of tools - the newer and more powerful the development tool, the smaller the pool of skilled professionals and the more unstable the tool functionality.
- Effectiveness of methods - what modeling, testing, version control, and design methods are going to be used, and how effective, efficient, and proven are they.
- Domain expertise - are skilled professionals available in the various domains, including business and technology.
- New features - what entirely new features are going to be added, and to what degree will these fit with current functionality.
- Methodology - does the overall approach to developing systems and using the selected methods promote flexibility, or is this a rigid, detailed approach that restricts flexibility.
- Competition - what will the competition do during the project? What new functionality will be announced or released.
- Time/Funding - how much time is available initially and as the project progresses? How much development funding is available.
- Other variables - any other factors that must be responded to during the project to ensure the success of the resulting, delivered system, such as reorganizations.

The overall complexity is a function of these variables:

complexity = f(development environment variables + target environment variables)

where these variables may and do change during the course of the project.

As the complexity of the project increases, the greater the need for controls, particularly the ongoing assessment and response to risk.

Attempts to model this development process have encountered the following problems:

- Many of the development processes are uncontrolled. The inputs and outputs are either unknown or loosely defined, the transformation process lacks necessary precision, and quality control is not defined. Testing processes are an example.

- An unknown number of development processes that bridge known but uncontrolled processes are unidentified. Detailed processes to ensure that a logical model contains adequate content to lead to a successful physical model are one such process.

- Environmental input (requirements) can only be taken into consideration at the beginning of the process. Complex change management procedures are required thereafter.

Attempts to impose a micro, or detailed, methodology model on the development process have not worked because the development process is still not completely defined. Acting as though the development process is defined and predictable, results in being unprepared for the unpredictable results.

Although the development process is incompletely defined and dynamic, numerous organizations have developed detailed development methodologies that include current development methods (structured, OO, etc.). The Waterfall methodology was one of the first such defined system development processes. A picture of the Waterfall methodology is shown in Figure 1.

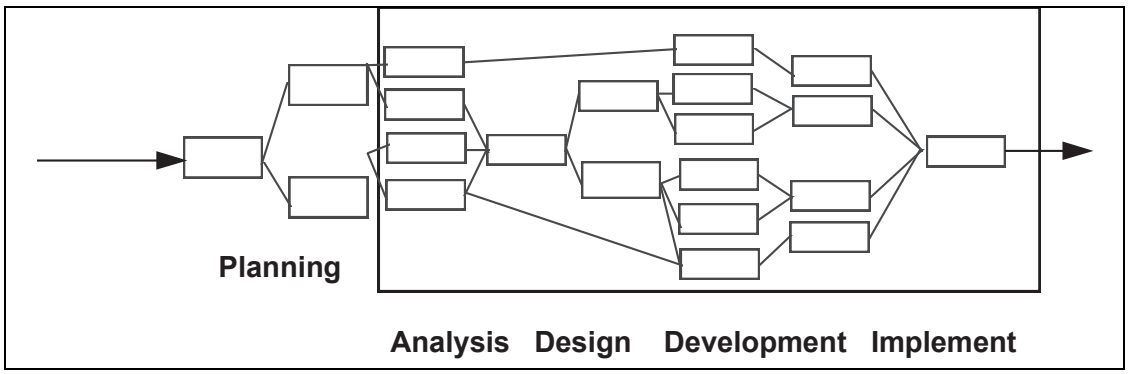


Figure 1 : Waterfall Methodology

Although the waterfall approach mandates the use of undefined processes, its linear nature has been its largest problem. The process does not define how to respond to unexpected output from any of the intermediate process.

Barry Boehm introduced a Spiral methodology to address this problem [50]. Each of the waterfall phases is ended with a risk assessment and prototyping activity. The Spiral methodology is shown in Figure 2.

The Spiral methodology "peels the onion," progressing through "layers" of the development process. A prototype lets users determine if the project is on track, should be sent back to prior phases, or should be ended. However, the phases and phase processes are still linear.

Requirements work is still performed in the requirements phase, design work in the design phase, and so forth, with each of the phases consisting of linear, explicitly defined processes.

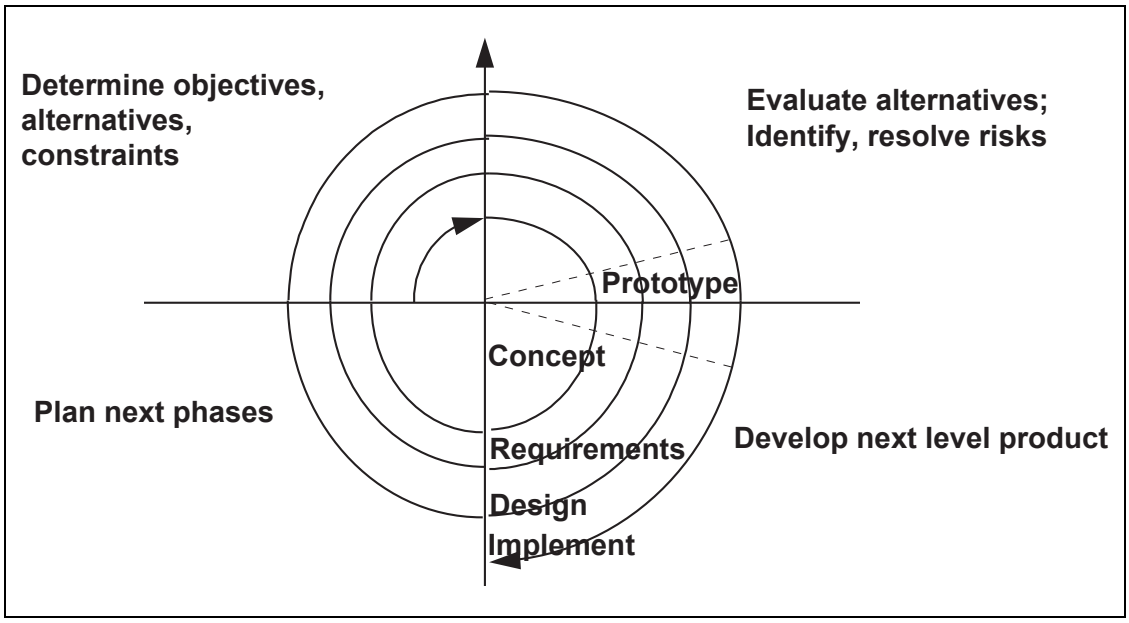


Figure 2 : Spiral Methodology

The Iterative methodology improves on the Spiral methodology. Each iteration consists of all of the standard Waterfall phases, but each iteration only addresses one set of parsed functionality. The overall project deliverable has been partitioned into prioritized subsystems, each with clean interfaces. Using this approach, one can test the feasibility of a subsystem and technology in the initial iterations. Further iterations can add resources to the project while ramping up the speed of delivery. This approach improves cost control, ensures delivery of systems (albeit subsystems), and improves overall flexibility. However, the Iterative approach still expects that the underlying development processes are defined and linear. See Figure 3.

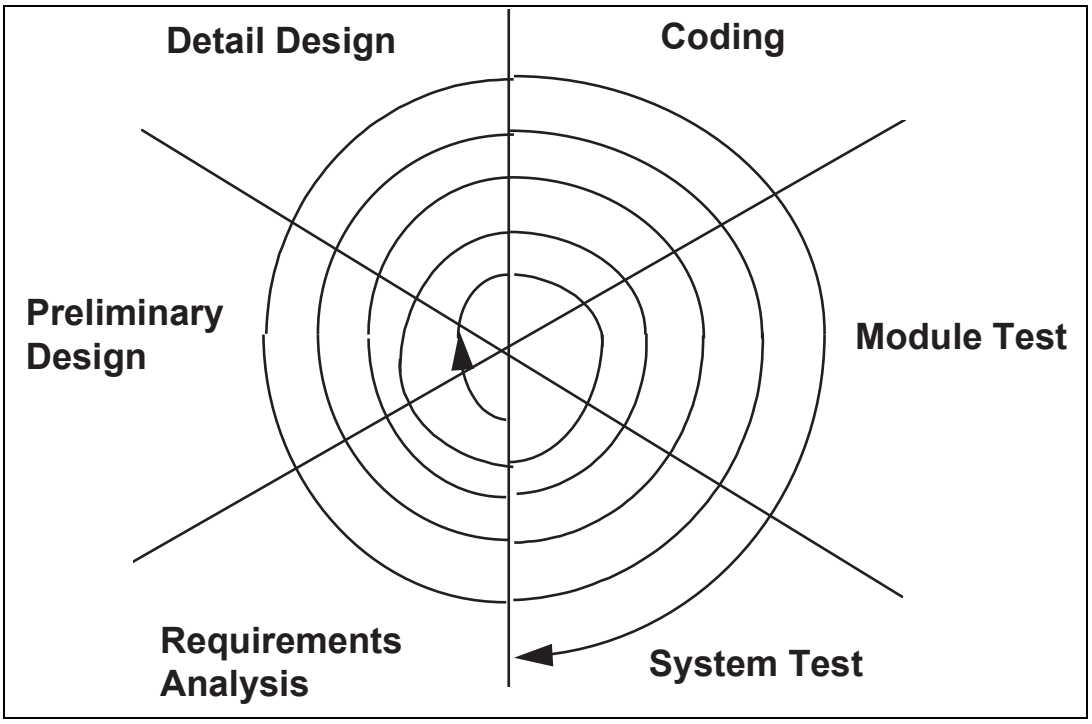


Figure 3 : Iterative Methodology

Given the complex environment and the increased reliance on new "state-of-the-art" systems, the risk endured by system development projects has increased and the search for mechanisms to handle this risk has intensified.

One can argue that current methodologies are better than nothing. Each improves on the other. The Spiral and Iterative approaches implant formal risk control mechanisms for dealing with unpredictable results. A framework for development is provided.

However, each rests on the fallacy that the development processes are defined, predictable processes. But unpredictable results occur throughout the projects. The rigor implied in the development processes stifles the flexibility needed to cope with the unpredictable results and respond to a complex environment.

Despite their widespread presence in the development community, our experience in the industry shows that people do not use the methodologies except as a macro process map, or for their detailed method descriptions.

The following graph demonstrates the current development environment, using any of the Waterfall, Spiral or Iterative processes. As the complexity of the variables increase even to a moderate level, the probability of a “successful” project quickly diminishes (a successful project is defined as a system that is useful when delivered). See Figure 4.

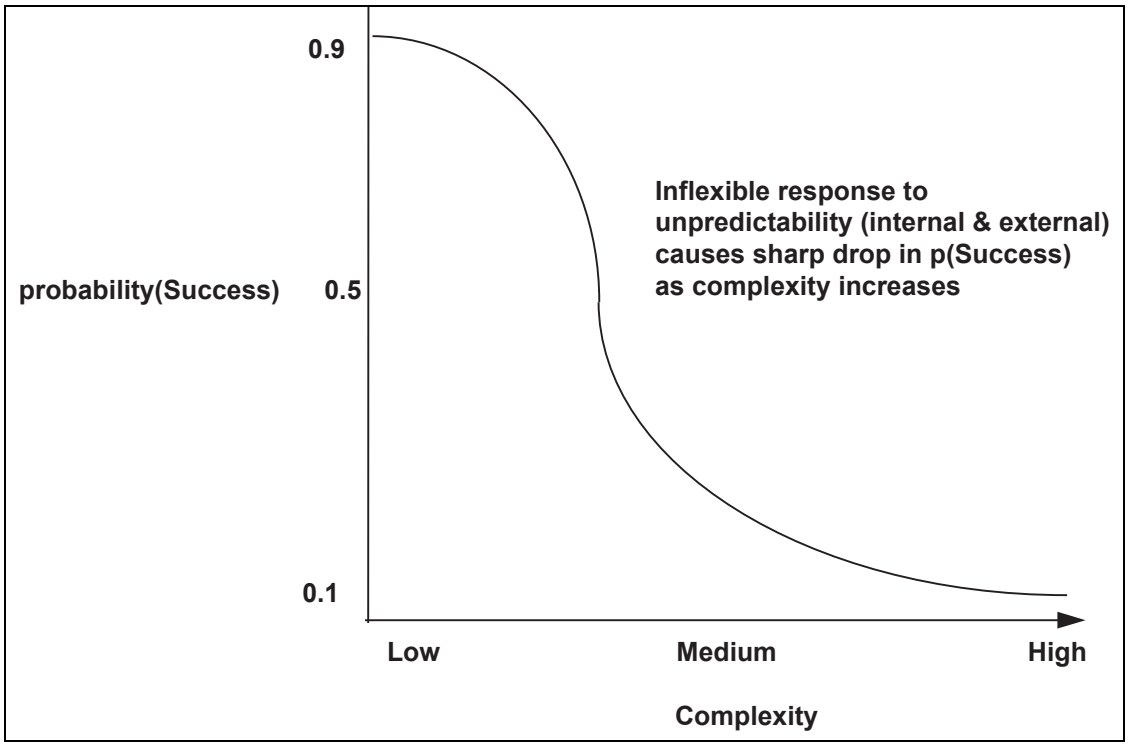


Figure 4: Defined Process Risk/Complexity Graph

4. Scrum Methodology

The system development process is complicated and complex. Therefore maximum flexibility and appropriate control is required. Evolution favors those that operate with maximum exposure to environmental change and have maximized flexibility. Evolution deselects those who have insulated themselves from environmental change and have minimized chaos and complexity in their environment.

An approach is needed that enables development teams to operate adaptively within a complex environment using imprecise processes. Complex system development occurs under rapidly changing circumstances. Producing orderly systems under chaotic circumstances requires maximum flexibility. The closer the development team operates to the edge of chaos, while still maintaining order, the more competitive and useful the resulting system will be. Langton has modeled this effect in computer simulations [6] and his work has provided this as a fundamental theorem in complexity theory.

Methodology may well be the most important factor in determining the probability of success. Methodologies that encourage and support flexibility have a high degree of tolerance for changes in other variables. With these methodologies, the development process is regarded as unpredictable at the onset, and control mechanisms are put in place to manage the unpredictability.

If we graph the relationship between environmental complexity and probability of success with a flexible methodology that incorporates controls and risk management, the tolerance for change is more durable. See Figure 5.

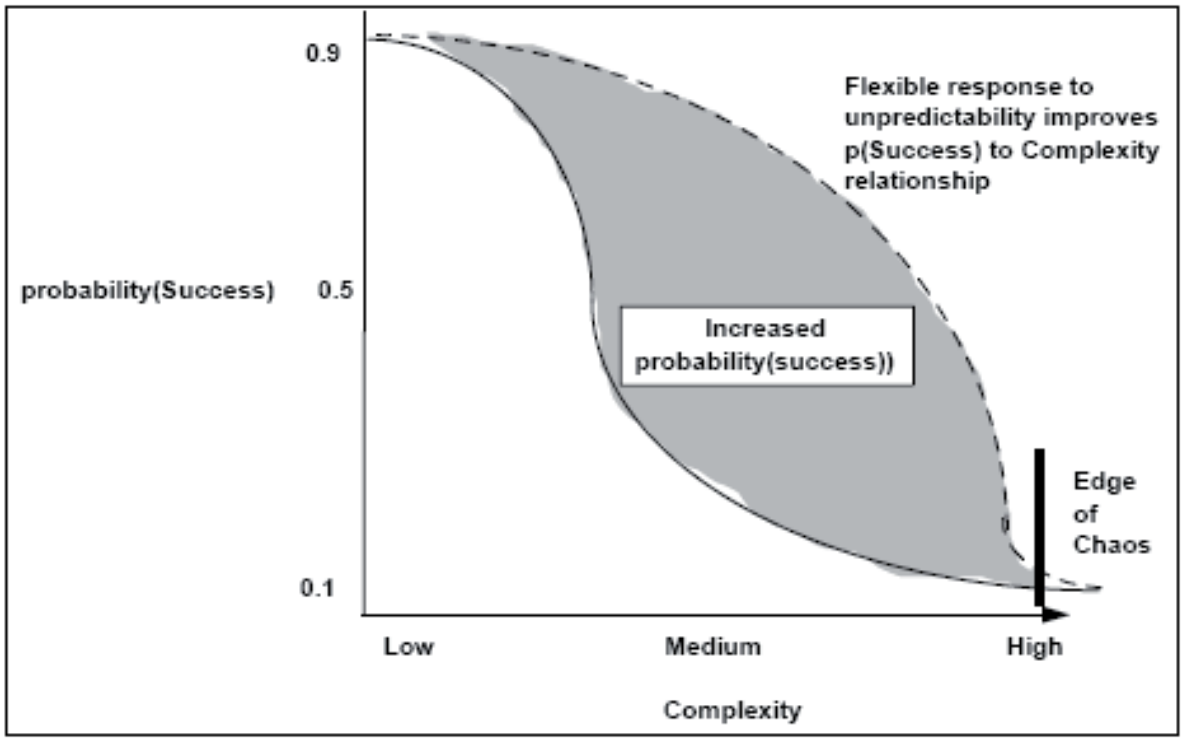


Figure 5: Risk/Complexity Comparison Graph

Figures 4 and 5 reflect software development experiences at ADM, Easel, VMARK, Borland and virtually every other developer of "packaged" software. These organizations have embraced risk and environmental complexity during development projects. Increased product impact, successful projects, and productivity gains were experienced. The best possible software is built.

Waterfall and Spiral methodologies set the context and deliverable definition at the start of a project. Scrum and Iterative methodologies initially plan the context and broad deliverable definition, and then evolve the deliverable during the project based on the environment. Scrum acknowledges that the underlying development processes are incompletely defined and uses control mechanisms to improve flexibility.

The primary difference between the defined (waterfall, spiral and iterative) and empirical (Scrum) approach is that the Scrum approach assumes that the analysis, design, and development processes in the Sprint phase are unpredictable. A control mechanism is used to manage the unpredictability and control the risk. Flexibility, responsiveness, and reliability are the results. See Figure 6.

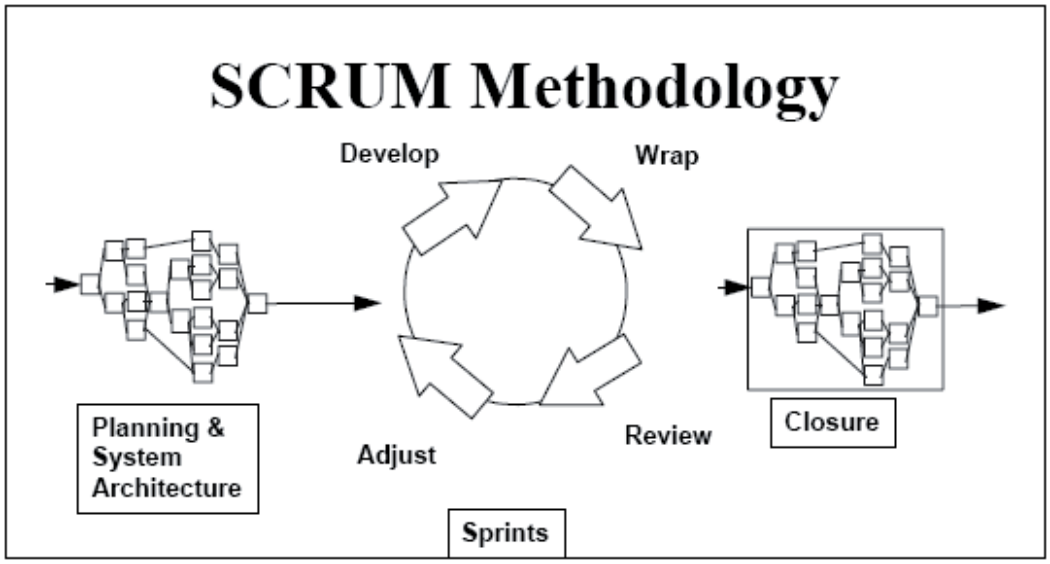


Figure 6 : Scrum Methodology

Characteristics of Scrum methodology are :

- The first and last phases (Planning and Closure) consist of defined processes, where all processes, inputs and outputs are well defined. The knowledge of how to do these processes is explicit. The flow is linear, with some iterations in the planning phase.

- The Sprint phase is an empirical process. Many of the processes in the sprint phase are unidentified or uncontrolled. It is treated as a black box that requires external controls. Accordingly, controls, including risk management, are put on each iteration of the Sprint phase to avoid chaos while maximizing flexibility.

- Sprints are nonlinear and flexible. Where available, explicit process knowledge is used; otherwise tacit knowledge and trial and error is used to build process knowledge. Sprints are used to evolve the final product.

- The project is open to the environment until the Closure phase. The deliverable can be changed at any time during the Planning and Sprint phases of the project. The project remains open to environmental complexity, including competitive, time, quality, and financial pressures, throughout these phases.

- The deliverable is determined during the project based on the environment.

The table in Figure 7 compares the primary Scrum characteristics to those of other methodologies.

	Waterfall	Spiral	Iterative	Scrum
Defined processes	Required	Required	Required	Planning & Closure only
Final product	Determined during planning	Determined during planning	Set during project	Set during project
Project cost	Determined during planning	Partially variable	Set during project	Set during project
Completion date	Determined during planning	Partially variable	Set during project	Set during project
Responsiveness to environment	Planning only	Planning primarily	At end of each iteration	**Throughout**
Team flexibility, creativity	Limited - cookbook approach	Limited - cookbook approach	Limited - cookbook approach	**Unlimited during iterations**
Knowledge transfer	Training prior to project	Training prior to project	Training prior to project	**Teamwork during project**
Probability of success	Low	Medium low	Medium	**High**

Figure 7 : Methodology Comparison

4.1 Scrum Phases

Scrum has the following groups of phases:

4.1.1. Pregame

- Planning : Definition of a new release based on currently known backlog, along with an estimate of its schedule and cost. If a new system is being developed, this phase consists of both conceptualization and analysis. If an existing system is being enhanced, this phase consists of limited analysis.
- Architecture : Design how the backlog items will be implemented. This phase includes system architecture modification and high level design.

4.1.2. Game

- Development Sprints : Development of new release functionality, with constant respect to the variables of time, requirements, quality, cost, and competition. Interaction with these variables defines the end of this phase. There are multiple, iterative development sprints, or cycles, that are used to evolve the system.

4.1.3. Postgame

- Closure : Preparation for release, including final documentation, pre-release staged testing, and release.

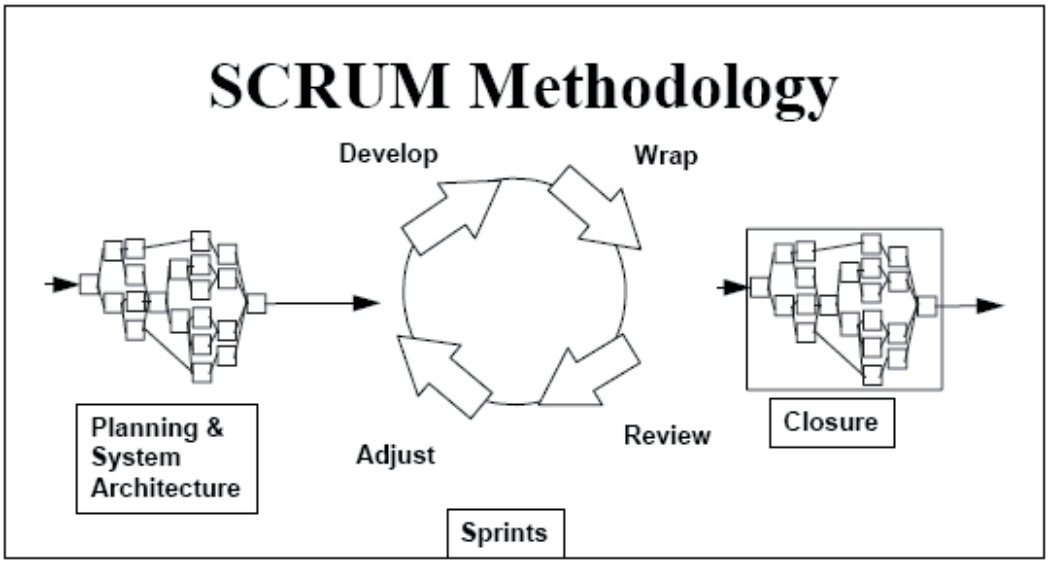


Figure 8: Scrum Methodology

4.2 Phase Steps

Each of the phases has the following steps:

4.2.1. Planning

- Development of a comprehensive backlog list.
- Definition of the delivery date and functionality of one or more releases.
- Selection of the release most appropriate for immediate development.
- Mapping of product packets (objects) for backlog items in the selected release.
- Definition of project team(s) for the building of the new release.
- Assessment of risk and appropriate risk controls.
- Review and possible adjustment of backlog items and packets.
- Validation or reselection of development tools and infrastructure.
- Estimation of release cost, including development, collateral material, marketing, training, and rollout.
- Verification of management approval and funding.

4.2.2. Architecture/High Level Design

- Review assigned backlog items.
- Identify changes necessary to implement backlog items.
- Perform domain analysis to the extent required to build, enhance, or update the domain models to reflect the new system context and requirements.
- Refine the system architecture to support the new context and requirements.

- Identify any problems or issues in developing or implementing the changes
- Design review meeting, each team presenting approach and changes to implement each backlog item. Reassign changes as required.

4.2.3. Development (Sprint)

The Development phase is an iterative cycle of development work. The management determines that time, competition, quality, or functionality are met, iterations are completed and the closure phase occurs. This approach is also known as Concurrent Engineering. Development consists of the following macro processes:

- Meeting with teams to review release plans.
- Distribution, review and adjustment of the standards with which the product will conform.
- Iterative Sprints, until the product is deemed ready for distribution.

A Sprint is a set of development activities conducted over a pre-defined period, usually one to four weeks. The interval is based on product complexity, risk assessment, and degree of oversight desired. Sprint speed and intensity are driven by the selected duration of the Sprint. Risk is assessed continuously and adequate risk controls and responses are put in place. Each Sprint consists of one or more teams performing the following:

- Develop: Defining changes needed for the implementation of backlog requirements into packets, opening the packets, performing domain analysis, designing, developing, implementing, testing, and documenting the changes. Development consists of the micro process of discovery, invention, and implementation.
- Wrap: Closing the packets, creating a executable version of changes and how they implement backlog requirements.
- Review: All teams meeting to present work and review progress, raising and resolving issues and problems, adding new backlog items. Risk is reviewed and appropriate responses defined.
- Adjust: Consolidating the information gathered from the review meeting into affected packets, including different look and feel and new properties.

Each Sprint is followed by a review, whose characteristics are :

- The whole team and product management are present and participate.
- The review can include customers, sales, marketing and others.
- Review covers functional, executable systems that encompass the objects assigned to that team and include the changes made to implement the backlog items.
- The way backlog items are implemented by changes may be changed based on the review.
- New backlog items may be introduced and assigned to teams as part of the review, changing the content and direction of deliverables.
- The time of the next review is determined based on progress and complexity. The Sprints usually have a duration of 1 to 4 weeks.

4.2.4. Closure

When the management team feels that the variables of time, competition, requirements, cost, and quality concur for a new release to occur, they declare the release "closed" and enter this phase. The closure phase prepares the developed product for general release. Integration, system test, user documentation, training material preparation, and marketing material preparation are among closure tasks.

4.3. Scrum Controls

Operating at the edge of chaos (unpredictability and complexity) requires management controls to avoid falling into chaos. The Scrum methodology embodies these general, loose controls, using OO techniques for the actual construction of deliverables.

Risk is the primary control. Risk assessment leads to changes in other controls and responses by the team.

Controls in the Scrum methodology are :

- Backlog: Product functionality requirements that are not adequately addressed by the current product release. Bugs, defects, customer requested enhancements, competitive product functionality, competitive edge functionality, and technology upgrades are backlog items.
- Release/Enhancement: backlog items that at a point in time represent a viable release based on the variables of requirements, time, quality, and competition.
- Packets: Product components or objects that must be changed to implement a backlog item into a new release.
- Changes: Changes that must occur to a packet to implement a backlog item.
- Problems: Technical problems that occur and must be solved to implement a change.
- Risks: risks that affect the success of the project are continuously assessed and responses planned. Other controls are affected as a result of risk assessment.
- Solutions: solutions to the problems and risks, often resulting in changes.
- Issues: Overall project and project issues that are not defined in terms of packets, changes and problems.

These controls are used in the various phases of Scrum. Management uses these controls to manage backlog. Teams use these controls to manage changes, problems. Both management

and teams jointly manage issues, risks, and solutions. These controls are reviewed, modified, and reconciled at every Sprint review meeting.

4.4 Scrum Deliverables

The delivered product is flexible. Its content is determined by environment variables, including time, competition, cost, or functionality. The deliverable determinants are market intelligence, customer contact, and the skill of developers. Frequent adjustments to deliverable content occur during the project in response to environment. The deliverable can be determined anytime during the project.

4.5 Scrum Project Team

The team that works on the new release includes full time developers and external parties who will be affected by the new release, such as marketing, sales, and customers. In traditional release processes, these latter groups are kept away from development teams for fear of over-complicating the process and providing "unnecessary" interference. The Scrum approach, however, welcomes and facilitates their controlled involvement at set intervals, as this increases the probability that release content and timing will be appropriate, useful, and marketable.

The following teams are formed for each new release:

Management: Led by the Product Manager, it defines initial content and timing of the release, then manages their evolution as the project progresses and variables change. Management deals with backlog, risk, and release content.

Development teams: Development teams are small, with each containing developers, documenters and quality control staff. One or more teams of between three and six people each are used. Each is assigned a set of packets (or objects), including all backlog items related to each packet. The team defines changes required to implement the backlog item in the packets, and manages all problems regarding the changes. Teams can be either functionally derived (assigned those packets that address specific sets of product functionality) or system derived (assigned unique layers of the system). The members of each team are selected based on their knowledge and expertise regarding sets of packets, or domain expertise.

4.6 Scrum Characteristics

The Scrum methodology is a metaphor for the game of Rugby. Rugby evolved from English football (soccer) under the intense pressure of the game:

> *Rugby student William Webb Ellis, 17, inaugurates a new game whose rules will be codified in 1839. Playing soccer for the 256-year-old college in East Warwickshire, Ellis sees that the clock is running out with his team behind so he scoops up the ball and runs with it in defiance of the rules.*
> The People's Chronology, Henry Holt and Company, Inc. Copyright © 1992.

Scrum projects have the following characteristics:

- Flexible deliverable - the content of the deliverable is dictated by the environment.
- Flexible schedule - the deliverable may be required sooner or later than initially planned.
- Small teams - each team has no more than 6 members. There may be multiple teams within a project.
- Frequent reviews - team progress is reviewed as frequently as environmental complexity and risk dictates (usually 1 to 4 week cycles). A functional executable must be prepared by each team for each review.
- Collaboration - intra and inter-collaboration is expected during the project.
- Object Oriented - each team will address a set of related objects, with clear interfaces and behavior.

The Scrum methodology shares many characteristics with the sport of Rugby:

- The context is set by playing field (environment) and Rugby rules (controls).
- The primary cycle is moving the ball forward.
- Rugby evolved from breaking soccer rules - adapting to the environment.

The game does not end until environment dictates (business need, competition, functionality, timetable).

5. Advantages of the Scrum Methodology

Traditional development methodologies are designed only to respond to the unpredictability of the external and development environments at the start of an enhancement cycle. Such newer approaches as the Boehm spiral methodology and its variants are still limited in their ability to respond to changing requirements once the project has started.

The Scrum methodology, on the other hand, is designed to be quite flexible throughout. It provides control mechanisms for planning a product release and then managing variables as the project progresses. This enables organizations to change the project and deliverables at any point in time, delivering the most appropriate release.

The Scrum methodology frees developers to devise the most ingenious solutions throughout the project, as learning occurs and the environment changes.

Small, collaborative teams of developers are able to share tacit knowledge about development processes. An excellent training environment for all parties is provided.

Object Oriented technology provides the basis for the Scrum methodology. Objects, or product features, offer a discrete and manageable environment. Procedural code, with its many and intertwined interfaces, is inappropriate for the Scrum methodology. Scrum may be selectively applied to procedural systems with clean interfaces and strong data orientation.

6. *Scrum Project Estimating*

Scrum projects can be estimated using standard function point estimating. However, it is advisable to estimate productivity at approximately twice the current metric. The estimate is only for starting purposes, however, since the overall timetable and cost are determined dynamically in response to the environmental factors.

Our observations have led us to conclude that Scrum projects have both velocity and acceleration. In terms of functions delivered, or backlog items completed:

- initial velocity and acceleration are low as infrastructure is built/modified
- as base functionality is put into objects, acceleration increases
- acceleration decreases and velocity remains sustainably high

Further development in metrics for empirical processes is required.

7. System Development Methodologies : Defined or Empirical

System development is the act of creating a logical construct that is implemented as logic and data on computers. The logical construct consists of inputs, processes, and outputs, both macro (whole construct) and micro (intermediate steps within whole construct). The whole is known as an implemented system.

Many artifacts are created while building the system. Artifacts may be used to guide thinking, check completeness, and create an audit trail. The artifacts consist of documents, models, programs, test cases, and other deliverables created prior to creating the implemented system. When available, a *metamodel* defines the semantic content of model artifacts. *Notation* describes the graphing and documentation conventions that are used to build the models.

The approach used to develop a system is known as a *method*. A *method* describes the activities involved in defining, building, and implementing a system; a *method* is a framework. Since a *method* is a logical process for constructing systems (process), it is known as a *metaprocess* (a process for modeling processes).

A *method* has micro and macro components. The macro components define the overall flow and time-sequenced framework for performing work. The micro components include *general design rules*, *patterns* and rules of *thumb*. *General design rules* state properties to achieve or to avoid in the design or general approaches to take while building a system. *Patterns* are solutions that can be applied to a type of development activity; they are solutions waiting for problems that occur during an activity in a method. **Rules of thumb** consist of a general body of hints and tips.

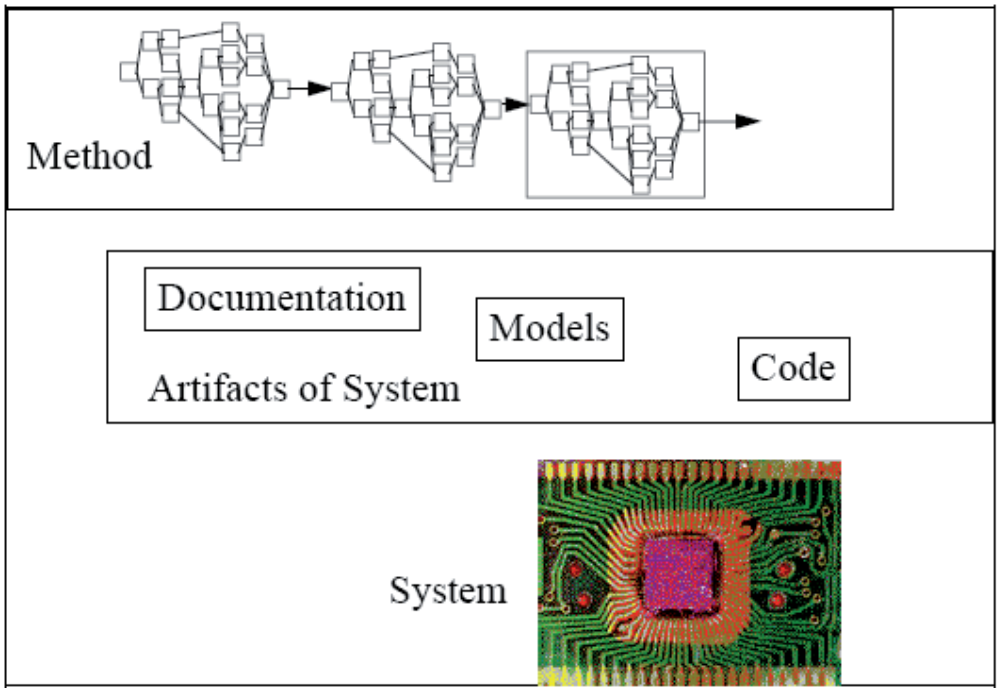


Figure 9: Relationship between the Method, the Artifacts, and the System

Applying concepts from industrial process control to the field of systems development, methods can be categorized as either "theoretical" (fully defined) or "empirical" (black box).

Correctly categorizing systems development methods is critical. The appropriate structure of a *method* for building a particular type of system depends on whether the *method* is theoretical or empirical.

Models of *theoretical processes* are derived from *first principles*, using material and energy balances and fundamental laws to determine the model. For a systems development *method* to be categorized as theoretical, it must conform to this definition.

Models of *empirical processes* are derived categorizing observed inputs and outputs, and defining controls that cause them to occur within prescribed bounds. Empirical process modeling involves constructing a process model strictly from experimentally obtained input/output data, with no recourse to any laws concerning the fundamental nature and properties of the system. No *a priori* knowledge about the process is necessary (although it can be helpful); a system is treated like a black box.

Primary characteristics of both theoretical and empirical modeling are detailed in Figure 10.

Theoretical Modeling	**Empirical Modeling**
1. Usually involves fewer measurements; requires experimentation only for the	Requires extensive measurements, because it relies entirely on experimentation for the

estimation of unknown model parameters.	model development.
2. Provides information about the internal state of the process.	Provides information only about that portion of the process which can be influenced by control action.
3. Promotes fundamental understanding of the internal workings of the process.	Treats the process like a "black box."
4. Requires fairly accurate and complete process knowledge.	Requires no such detailed knowledge; only that output data be obtainable in response to input changes.
5. Not particularly useful for poorly understood and/or complex processes.	Quite often proves to be the only alternative for modeling the behavior of poorly understood and/or complex processes.
6. Naturally produces both linear and nonlinear process models.	Requires special methods to produce nonlinear models.

Figure 10: Theoretical vs. empirical modeling

Upon inspection, we assert that the systems development process is empirical:

- Applicable first principles are not present
- The process is only beginning to be understood
- The process is complex
- The process is changing

Most methodologists agree with this assertion; "...you can't expect a *method* to tell you everything to do. Writing software is a creative process, like painting or writing or architecture... ... (*a method*) supplies a framework that tells how to go about it and identifies the places where creativity is needed. But you still have to supply the creativity...."[51]

Categorizing the systems development methods as empirical is critical to the effective management of the systems development process.

If systems development methods are categorized as empirical, measurements and controls are required because it is understood that the inner workings of the method are so loosely defined that they cannot be counted on to operate predictably.

In the past, methods have been provided and applied as though they were theoretical. As a consequence, measurements were not relied upon and controls dependent upon the measurements weren't used.

Many of the problems in developing systems have occurred because of this incorrect categorization. When a black box process is treated as a fully defined process, unpredictable results occur. Also, the controls are not in place to measure and respond to the unpredictability.

Artikel 9 bij leereenheid 3

Scrum Anti-patterns – An Empirical Study
Veli-Pekka Eloranta, Kai Koskimies,Tommi Mikkonen en Jyri Vuorinen

20th Asia-Pacific Software Engineering Conference, 2013

2013 20th Asia-Pacific Software Engineering Conference

Scrum Anti-patterns – An Empirical Study

Veli-Pekka Eloranta, Kai Koskimies,Tommi Mikkonen and Jyri Vuorinen
Department of Pervasive Computing,
Tampere University of Technology
Tampere, Finland
{firstname.lastname}@tut.fi

***Abstract*—Wide-spread adoption of the agile movement has rapidly changed the landscape of software industry. In particular, Scrum is an agile process framework that has become extremely popular in industry. However, the practical implementation of Scrum in companies rarely follows the text book ideals. Typically, companies deviate from the proposed Scrum practices for different reasons. While some deviations may be well motivated and reasonable, companies are often tempted to adjust Scrum for the company without clearly understanding the consequences of the deviations. In this paper our aim is to identify ways of potentially harmful mishandling of Scrum in industry based on empirical data collected in a survey. The identified (mal)practices are presented in a semi-formal manner as anti-patterns. The study resulted in 10 anti-patterns that express the context of the deviation, the deviation itself, the broken core principles of Scrum, and the possible consequences of the deviation.**

Keywords—Agile software development, Scrum, anti-patterns, Scrum problem areas

I. INTRODUCTION

The wide-spread adoption of the agile movement – culminating in many ways in the Agile Manifesto published in 2001 [1] at the level of ideals – has taken off in the software industry in a rapid pace. Within a relatively short period, an increasing number of organizations have adopted an agile way of working in their R&D departments. However, when inspected more closely and in more detail, by phrase "we are agile in software development", companies often mean simply, "we are using some practices or ideas of Scrum". This is often referred as ScrumBut [14].

Scrum is a simple, iterative framework for project management [12, 13, 15]. At present, the adoption of Scrum has progressed in several organizations so far that it is possible to start collecting empirical data on the use of Scrum. To consolidate accurate, practical knowledge that is relevant to companies developing software, it is crucial to gather data on the Scrum practices in action for determining which parts of the methodology pose problems in industry. In this paper, our goal is to identify empirically the problem areas in Scrum and practices that are hard to adopt in the organizations, and to present the found Scrum pitfalls in a systematic manner as so-called anti-patterns [2]. Anti-patterns are common practices that are seen initially convenient and appropriate, but that are actually harmful in the long run and should therefore be avoided. An anti-pattern is often associated also with an alternative, recommended practice, which is more appropriate in most of the cases.

In principle, there are two kinds of problems in adopting Scrum: (1) harmful deviations from recommended Scrum practices and (2) recommended Scrum practices that are for some reason unsuitable in a particular context. Both types of problems are important to be recognized. For the first type, a company starting to use Scrum should be aware of common deviations that may look reasonable but that are actually harmful. For the second type, understanding well-motivated deviations from the standard, text-book Scrum provides information for refining the methodology to fit the purposes of companies developing software. In this paper we will not distinguish between these two types of Scrum problems. The rationale for this decision is that even though a Scrum deviation may be well-motivated in a particular context, the deviation has harmful consequences in most cases. Instead, we discuss in each anti-pattern the possible exceptions when the Scrum deviation might be sensible. In general, the alternative "good" practice which is part of an anti-pattern is the recommended text book Scrum practice.

To mine typical Scrum pitfalls in industry, we have conducted a survey of Scrum related practices in IT companies that use Scrum. The results of the survey are presented in this paper in terms of basic Scrum practices. For each basic practice, we summarize the contents of the interviews, especially concentrating on the ways the practices have been realized in the companies and possible deviations from Scrum recommendations. Then, we formulate the identified deviations from Scrum as anti-patterns using a format specifically designed for this purpose. These anti-patterns, together with the empirical data supporting them, are the main contribution of this paper.

The rest of this paper is structured as follows. Section II provides the basic information concerning the survey method. In Section III we summarize the results of the survey structured according to the basic Scrum concepts, discussing the extent to which companies follow the related Scrum practice and the possible deviations. Section IV condenses the contribution of this paper by formulating the identified problematic practices as anti-patterns. Section V discusses the limitations of this study, and Section VI summarizes previous work on the usage of Scrum in industry, with comparisons to our work. Towards the end of the paper, Section VII draws final conclusions and introduces directions for future work. We assume familiarity with Scrum throughout the paper.

II. SURVEY METHOD

Survey Research [6], commonly used to assess opinions and feelings, was selected as a research method for this study. The method gives an opportunity to identify characteristics of a group of individuals, and based on the results, one can compare the views of different populations.

The interview covered 11 IT companies in Tampere region, which is one of the centers of IT industry in Finland. The representatives of selected companies had participated in a certified Scrum master course. The survey is based on structured interviews of persons that play (or have played at the time) a major role in software development organizations using Scrum. Companies participating in the survey are of mixed background, some working for customer projects, and some acting in a more product-driven fashion. The range of systems includes embedded devices as well as more traditional applications, such as web-based software.

In total, 18 representatives of teams were interviewed, including project, development and quality managers as well as developers. A typical title of a person being interviewed for the survey was company development manager, who can provide as wide and covering information regarding company's software development process as possible. In addition, since the majority of the interviewed persons had senior positions, they could also provide information on how Scrum was integrated with other activities of the company.

There were 48 questions in the survey. They were composed so that the answers would cover all the objectives of this study. In the interview, additional questions were asked concerning Scrum basics whenever necessary to find out if the organization has deviated from the practice and adopted their own way of working or if the organization had not taken the practice in use in the first place. The complete list of questions can be found at [7].

III. SURVEY RESULTS

A. Sprints

Sprint is a time-boxed period where the actual work in Scrum takes place. The original Scrum paper [12] does not define any length for the sprint, but among the practitioners the length of 2 to 4 weeks has become a de facto practice. Scrum guide [15] also advices to use 2 to 4 week sprints. The results of the question on sprint length are shown in Table I.

Table I. Sprint lengths in the interviewed teams

< 2 weeks	2 weeks	3 weeks	4 weeks	> 4 weeks	Varying length	No sprints
1	9	3	2	1	1	1

As Table I indicates the most widely used sprint length is 2 weeks. One company reported having a varying sprint length meaning that if the work takes more time than planned then the sprint is extended. Furthermore, one company reported that they did not use sprints at all, even though they claimed that they used Scrum.

Most of the teams had experimented with the sprint length. The most typical scenario was that the team had started with 4 week sprints and noticed that it is too long. In most such cases 4 weeks was too long as the team could not respond to customer requests and the feedback loop became too long. In addition, two teams reported that when they had used 4 week sprints, only two thirds of the sprint's work was done. When they shortened the sprint length, they could achieve situation where all tasks were completed at the end of the sprint.

Roughly half of the teams had tried sprint lengths shorter than two weeks, most typically 1 week or 1.5 weeks. However, they had abandoned short sprints as the management overhead became too large. Sprint planning and sprint reviews was taking too much of the sprint length. Furthermore, typically two weeks were short enough time to be able to satisfy customer's change requests.

One team had two overlapping iterations at the same time. They had iterations of 4 weeks and 2 week sprints within the iteration. They had to take this approach as the customer did not want to participate in meetings every two weeks. Some of the teams had sprint length of 2 to 4 weeks, but they had separate implementation and testing sprints meaning that every other sprint they implemented new features and the next sprint was for testing those features. Of course, this approach does not produce potentially shippable product increment each sprint.

B. Testing

In Scrum, every sprint should produce a potentially shippable product increment. In order to accomplish that, the software has to be tested before the sprint ends. Typically this is incorporated in the definition of "done". In the interview, we asked the companies to what extent the software is tested within a sprint. Results are shown in Table II.

Table II. State of the testing of software after each sprint

Not tested	Unit tested	Integr. tested	System tested	Acceptance tested	Delivered and installed
3	8	2	2	2	1

Eight teams had the software unit tested at the end of the sprint. However, two of those teams said that they had efforts going on to make the testing more automated and the goal was to achieve automated system testing. In some cases, the system testing was carried out always in the next sprint by a separate team. This is, however, problematic as new functionality might be already written on top of the old code which is still in the testing phase. Only three teams had the code acceptance tested within the sprint. One of these teams was also deploying the product within the sprint. However, they had had to limit the deployment to every other sprint as the customer did not want to update the software so often.

According to the interviewed teams, the biggest obstacle on the road to test the code more within the sprint was "waterfallish" thinking where testing is carried out at the end

of the sprint and the lack of automation in the testing. In addition, some of the interviewed companies were manufacturers of large machines and in these systems testing is even harder as one should test the software on the hardware before it can be shipped.

C. Specifications

In Scrum it is advised to document requirements as user stories or use cases in the product backlog. Table III shows the methods interviewed companies used to make specifications for their products.

Table III. Specifications documentation

Big requirement documents	User stories	Use cases	Face-to-face	Feature descriptions
4	8	3	1	2

As the results show, four teams were still using traditional big requirement documents for specifications. Most of the teams (8) were using user stories. Two of these teams reported that they have separate UI specifications or more detailed specifications when necessary to accompany user stories. Three teams used use cases. One of these teams said that they prefer use cases over user stories as use cases document context for the feature and provide some rationale for the functionality. One team had face-to-face discussions where the product was specified. After discussions with the customer and the product owner they documented just enough specifications that they needed to remember what was discussed. Two teams used only short feature descriptions in their product backlog.

As stated by Cockburn [3], the most efficient way of communicating is face-to-face. It would be advisable to have face-to-face discussions especially between the customer and the development team when making the specifications. If the requirements are communicated using written user stories, the communication is one-directional and not efficient as the development team cannot ask questions from the customer.

D. Product owner

Product owner is responsible for maximizing the value of the product [15]. Product owner is also responsible for organizing the product backlog and creating enabling specifications for the product owner team. Table IV shows the distribution of the different approaches with respect to the product owner.

Surprisingly, altogether 7 teams reported not to have a product owner at all. Furthermore, 3 teams had a product owner who was customer representative. This is a bit alarming as the product owner is responsible for optimizing ROI. If the customer is in a position to optimize ROI for the company, it cannot be beneficial for the company in the long run. One team reported to have a shared product owner. This means that the company had one product owner who was managing more than one product. It was not very surprising that this team had problems as the product owner did not have enough time to create enabling specifications.

Table IV. Product owner role

No product owner	Own product owner	Product owner from customer	Shared product owner
7	7	3	1

One team also reported to have product owner team, i.e. there are multiple product owners that work as a team. They had evolved to this practise from the single product owner as the teams had been complaining that the single product owner did not have enough time to address the questions the teams had.

One interesting point that was discovered multiple times in the interviews was that teams who did not have product owner and were working on customer projects said that having a product owner would not work for them. They argued that if they were building their own product, it might work better. On the other hand, multiple teams building their own product and having no product owner reported that the product owner role would work for them only if they were doing customer projects. So independently of the project type, the teams found the product owner role to be problematic.

E. Product backlog

Product backlog is an ordered list of things that might be needed in the product and acts as a single source of requirements for the product [15]. The product backlog should be ordered so that the most valuable, highest-risk or priority items are on the top (taking dependencies into account). These top items are also the most analyzed, clear and ready for development. Table V shows the usage of product backlog in the interviewed teams.

Table V. Usage of Product backlog

No Product backlog	Product backlog	Ordered product backlog	Multiple product backlogs
3	7	7	1

As can be seen from Table V, most of the teams were using product backlog. Only three teams were not using it. None of these three teams reported that they have evolved away from this practise; they just had not adopted it. The largest observed problem with the product backlog was that in roughly half of the teams, the product backlog was not ordered. When considering the results of the product owner questions (7 teams not having a product owner) is not a big surprise that the product backlog was not ordered in so many companies. The interviewed teams reported that the main obstacle in ordering the backlog was that the product owner did not have enough time for it or product owner did not have required competence to order the list. However, in the interviews we also discovered that some teams were ordering the product backlog with the product owner as he was not able to do it alone. Only one team reported that they work for more than one product owner and select items from multiple backlogs to the sprint.

F. Estimates

Work estimation is an important aspect of Scrum, enabling the efficient use of resources. In the survey we asked who is responsible for making such estimates, if anyone. We also investigated how many of the teams used hours for estimation and how many used so called estimation points. The results concerning the responsible party of the estimates are shown in Table VI.

Table VI. Who estimates the PBIs

No estimates	Project manager / PO estimates	Team estimates
2	6	10

Two of the interviewed teams did not do estimates at all. For 6 teams, a project manager or a product owner made the estimates. This number is surprisingly big and indicates that the project manager or the product owner dictates which items are implemented in which sprint. Later on the question about the self-organization confirmed this conclusion. In total, ten teams produced the work estimates by themselves.

Out of the 16 teams having estimates, eight created their estimates using hours. On the other hand, eight used estimation points or other imaginary unit to estimate the work amount. Planning poker was used by five teams out of the eight applying estimation points.

G. Sprint burn-down charts and velocity

Sprint burn-downs are used to illustrate remaining work in sprint or in some cases in release [15]. Generally the idea is to burn the chart down when a task is completed. However, different variations are often used.

In total seven teams did not use burn-downs whereas eleven teams used them. However, in interviews one team said that they have it in use, but they have never seen it. The project manager was drawing it for himself to get estimates concerning the possible failure of the sprint. Seven teams did not use burn-downs. The main reason for not using them was that they did not see any benefits gained out of drawing the chart – it is just extra work to maintain the chart. For some teams a reason was just that there was no suitable place in the wall for the chart.

Out of the eleven teams who used burn-down charts, eight had partial burn-downs meaning that they used hours in the burn-down chart and once they had worked on some task, they immediately removed it from the remaining work. Only three teams burned down the graph when the task was completed.

Additionally, we asked if teams know their velocity, meaning how many work items they can do within a sprint. Eight teams used velocity to estimate the amount of work items they can deliver in sprints. Ten teams did not know their velocity.

H. Team disruption

Generally, Scrum advices to protect the team from outside disruptions. However, in practice this might be hard to achieve as team members have to answer phone calls, take bug reports and maintain old projects. Therefore, it is not very surprising that 13 of the interviewed teams were disrupted during the sprint and only five teams were not. In one case, it was the project manager disrupting the team, in other cases, typical cause of disruption was the customer of the current (or old) project.

I. Self-organization and cross-functionality of the team

In Scrum, it is critical that teams are cross-functional meaning that they have all the necessary expertise to deliver working software. Furthermore, the team should be self-organized so that each team member can choose what to work on and there is no one assigning tasks. Therefore, we asked if the companies had cross-functional teams and are they self-organizing. Ten teams had the necessary skills (e.g. design, coding and testing) to deliver a working product. In eight cases there were, for example, separate testing teams and therefore the team was not cross-functional. This usually causes problems, a typical one being that the product is not ready to be delivered after the sprint as it still needs to be tested. Furthermore, testing typically takes place during the next sprint and, consequently, new code has often been written on top of the code being tested at the same time.

The last question about the Scrum practices concerned the self-organization of the team, that is, to what extent the team gets to decide what they are working on. Half of the interviewed teams, i.e. nine, were self-organized and the other half had a project manager to assign tasks to certain persons.

IV. Identified Scrum Anti-patterns

A. Anti-pattern format

We will use the format presented in Table VII for the presentation of the anti-patterns. The used anti-pattern format is derived from [2] with small modifications to better suit the organizational nature of the presented anti-patterns. The Scrum recommendation is taken from Scrum literature, for example [15]. The context is extracted from the results of the interview, reflecting the explanations the interviewees gave for the Scrum deviations. The consequences are partly taken from the interviews as the problems noted by the interviewees and partly from Scrum literature, mostly [15]. Often the existence of another anti-pattern was observed to be one of the causes of an anti-pattern. Exceptions are situations showing up in the interviews that were considered to be "justified Scrum deviations" by the authors. In some cases, no notable exceptions could be identified.

B. List of identified anti-patterns

Ten anti-patterns were identified based on the survey, as presented in Table VIII. The following criteria were used to accept a practice as an anti-pattern: 1) the practice is a deviation of Scrum recommendations, 2) the practice was observed in more than one team, and 3) the practice has

unfavourable consequences noted explicitly either by the interviewed team or by Scrum literature.

Table VII. Anti-pattern format

Anti-pattern field	Contents
Name	The name of the anti-pattern
Scrum recommendation	The related Scrum recommendation
Context	Specific conditions under which the anti-pattern typically occurs
Solution	The anti-pattern (usually harmful) solution or Scrum deviation
Consequences	The consequences of using the anti-pattern
Exceptions	Possible situations in which the use of the anti-pattern could be justified

Table VIII. Identified anti-patterns

Name: **Too long sprint** *Scrum recommendation*: 2 weeks sprints. *Context*: Early experimenting with Scrum, waterfall legacy of long production cycles, customers unwilling to participate in short intervals. *Solution*: Using 4 weeks or even longer sprints. *Consequences*: In spite of longer working time for sprints, the planned tasks tend to be unfinished at the end of sprint, possibly because the first weeks are not used efficiently enough and too large tasks are allocated to the sprints. Consequently diminishes team commitment to deliver the items at the end of the sprint. Feedback cycle becomes so long that some of the work might not be needed anymore. *Exceptions*: If the development must be synchronized with external work that has slower pace (e.g. hardware development), longer sprints may be justified.
Name: **Testing in next sprint** *Scrum recommendation*: All testing of software is done within the sprint that creates the software, to enable the shipping of a working product increment. *Context*: Waterfallish thinking of the team, separate testing team, lack of testing automation, too short sprint anti-pattern, too long sprint anti-pattern. *Solution*: Testing is done in the next sprint following a development sprint. *Consequences*: New code may be written on top of non-tested code. The product is not always in a "potentially shippable" state. Bugs need to be fixed long after the code is written. As the bug is found after several weeks from implementing the feature, it might require some time to recall how the feature was implemented. Increases cognitive load. *Exceptions*: In an embedded system, it may be necessary to test the code together with real hardware, which might be available only at the later stage of the development.
Name: **Big requirements documentation** *Scrum recommendation*: Product backlog contains all potential items that are going to be implemented and this list is ordered by value, risk, or dependencies, for instance. Items might be described as user stories or use cases and so on. *Context*: Waterfallish thinking of the team, legacy company guidelines for requirements documentation. *Solution*: Traditional big requirements documentation is produced prior to the development sprints. *Consequences*: Requirements are not well understood by the team. Requirements are not ordered by the value or other criteria. Thus, it is harder to decide which items should be implemented next. *Exceptions*: In safety-critical systems, various regulations enforce the existence of large requirements documentation and traceability to the code.
Name: **Customer product owner** *Scrum recommendation*: Product owner role is necessary to ensure that the product produces value to the customer. *Context*: Insufficient understanding of the role of a product owner, lack of persons suitable for that role, customer does not trust the development team. *Solution*: A customer representative acting as a product owner. *Consequences*: Requirements are not well understood by the team, leads to Customer caused disruption anti-pattern Customer tries to add new items to the sprint during the sprint. Customer optimizing the return on investment for the company. *Exceptions*: If suitable person to represent customer is not available, Product owner surrogate could be used temporarily.
Name: **Product owner without authority** *Scrum recommendation*: Product owner is responsible for managing Product backlog and deciding which items maximize the value of the product. *Context*: Insufficient understanding of the role of a product owner, suitable persons are not motivated to use Scrum, general lack of interest, large organizations where responsibility over products and development may be fragmented. *Solution*: A development team member is promoted as product owner as product managers are not interested in using Scrum. *Consequences*: Product owner does not have the authority to decide on which items are implemented and which are discarded. Product owner can not really make decisions concerning the product. There might be other managers to above product owner to make these decisions. Product owner is not necessarily in direct contact with the customer. *Exceptions*: If product owner with authority is not temporarily available, a surrogate product owner could be used for a couple of sprints.

Name: **Unordered product backlog**
Scrum recommendation: Product backlog is ordered to give precedence for high-risk or the most valuable items.
Context: Product owner without authority anti-pattern, Customer product owner anti-pattern, insufficient competence of the product owner.
Solution: Product backlog is not ordered, but teams select items based on their own judgement.
Consequences: Lacking vision of the risky or valuable elements of the product. Building wrong features. Only the nice features get implemented. Features being hard to implement or test are left to the backlog.
Exceptions: Projects where the requirements are fixed and given up-front by the customer. In addition these requirements are not going to change. Some initiatives by governments and military might have such requirements.

Name: **Work estimates given to teams**
Scrum recommendation: Teams should produce work estimates for themselves.
Context: Hierarchical working practices of the company. Need to know an estimate in advance on how much the product will cost.
Solution: Product manager or product owner produces work estimates.
Consequences: Lacking commitment of the team. Wrong estimates: team is better to estimate than a single person who is not going to implement the task. Out-dated estimates as items have dependencies.
Exceptions:

Name: **Invisible progress**
Scrum recommendation: The progress of the work should be made visible with burn-down charts.
Context: Hierarchical working practices of the company, unsuitably distributed team
Solution: Product manager produces burn-down chart for herself, or no visual representation of the progress is produced. Sometimes partially completed tasks are marked as finished.
Consequences: Lacking progress awareness of the team, risk of failing to deliver all features in a sprint. Problems can be hidden. Scrum Master's work becomes harder.
Exceptions: Very small projects where people know each other, share a room, and can discuss the progress.

Name: **Customer caused disruption**
Scrum recommendation: Teams should be protected against any outside disruptions.
Context: Improper product owner anti-pattern, close and interactive co-operation with a customer.
Solution: Customers are allowed to communicate directly with team members and they can negotiate with the team to add new features to the sprint
Consequences: Work flow of the team is interrupted, reducing the efficiency of the team. New features may crawl in to the product without Product owner's involvement. The value stream is compromised.
Exceptions:

Name: **Semi-functional teams**
Scrum recommendation: Teams should be cross-functional, providing all the necessary skills to carry out the full realization of shippable products.
Context: Testing in next sprint anti-pattern, lacking expertise in the company staff, division to different teams according to the expertise.
Solution: Team produces only certain aspect of the shippable product.
Consequences: Divided responsibility of producing a shippable product, less committed team. Extraneous need for cross-team communication. Slower feedback loops.
Exceptions: Highly specialized and demanding software architecture design carried out before the sprints.

C. *Scrum anti-pattern usage and company characteristics*

To investigate the possible effect of the characteristics of the company on the tendency to resort to Scrum anti-patterns, we divided the companies into three categories according to their size: small (less than 30 developers), medium (30-100 developers), and large (over 100 developers). In addition, the companies were also categorized according to years of Scrum experience: less than a year, 1 to 2 years, and over 2 years. Table IX shows the results.

Two significant observations can be made from the results. First, it seems that smaller companies follow Scrum principles more closely, avoiding anti-patterns. This might be due to fact that large organizations tend to have legacy practices which prevent further adoption of new practices. Lindvall et al. [8] also states that in large organizations, new practices may result in double work as old practices need to be followed when new practices are taken into use.

The second observation from the result is that companies having more Scrum experience tend to use anti-patterns more. On one hand this can be explained by the fact that companies might have changed their way of working as a part of continuous improvement. On the other hand, if this is the case, they have adopted anti-patterns as a part of process improvement, which, of course, is not the intention. This might need further research to find out the reasons behind the anti-pattern adoption in these companies and what kind of consequences that might have.

Two anti-patterns seem to be common despite of company size and Scrum experience: Testing in the next sprint and Customer caused disruption. The popularity of the first anti-pattern indicates that the transformation to agile methods is particularly challenging in testing. The frequency of the second anti-pattern hints that companies do not want that the used development process affects their customer relationships and therefore let the customer disrupt the development teams.

V. LIMITATIONS

The survey was carried out in a limited region, which may influence the results. However, as different kinds of companies were included in the survey, we believe that the results can be generalized. In Survey Research, the question

design is critical to get useful and valid data [6]. In this survey, the main questions, however, are directly related to Scrum basic concepts and if the interviewee understand Scrum, the questions are likely to produce useful data. Therefore, question design is not a major threat to the validity of this survey.

Table IX. Summary of Scrum anti-patterns in different companies

Name	Company size			Scrum Experience		
	S	M	L	<1	<2	2+
Too long sprint		2	2	1		3
Testing in next sprint	1	1	4	1	1	4
Big requirements document		1	3	1		3
Customer product owner	1	1	2		2	2
Product owner without authority		2	3	1		4
Unordered product backlog	1	1	3	1	1	3
Work estimates given to teams		1	1	1		1
Invisible progress	1		3		1	3
Customer caused disruption	2	2	4	1	3	4
Semi-functional teams		1	2	1		2

Another limiting factor is that the number of interviewed companies was relatively small, which might lead to biased sample [6]. Conducting this kind of interview study for a large number of companies would be difficult, not only because of the required amount of resources but also because companies are not necessarily willing to share information about their problems, and even use some critical resources to do it.

Despite the above limitations, the results of this survey are in line with other survey-based studies, as will be discussed next. Still, it is likely that other anti-patterns would be probably found in other surveys, which does not invalidate the findings here.

VI. Related Work

As far as we know, there is no prior empirical work on identifying Scrum anti-patterns. In contrast, numerous positive properties have been associated with agile development approaches. Team communication – giving and receiving frequent feedback – is often reported to have increased when adopting agile methods in a company (see e.g. [10]). Also the regular Team meetings have been reported to support the collaboration and to give everyone better general view of the work progress. Customers are seen as valuable resources, and the customers themselves experience more active participation in the process – in contrast to the one of the weaknesses of the traditional waterfall approach where customers' current needs are not addressed until later on in the project [4]. Further benefits of agile approaches also include better process control, transparency, and improved product quality because of practices such as continuous integration and controllable-sized tasks [3]. These were confirmed in our interviews.

A recent study on agile methods by VersionOne gathered data from over 4000 survey participants. The study shows that Scrum (including also different kinds of Scrum hybrids) has 72% market share [16]. The study also lists the main benefits that companies reported being gained from the use of agile methods. The benefits include the following: 1) better control on managing changing priorities, 2) improved visibility and team morale, 3) quicker time-to-market and 4) increased productivity. The study also lists that the leading causes for failed agile projects are the following: 1) lack of experience in agile methods, 2) the organization culture does not support agile ways of working, and 3) external pressure to use traditional waterfall practices. All these problem areas were also identified in our interviews. In addition, the report lists barriers to further adoption of agile practices. The most significant barriers are: 1) lacking ability to change organization culture, 2) general resistance to change, 3) availability of skilled personnel, and 4) lack of management support. These all are also in line with our results from the interviews, giving further confidence towards the validity of our results. However since our results have been gathered with personal interviews, we believe the results have more depth than what can be collected with questionnaires only. In particular, the potential problems and deviations are easier to identify in personal interviews.

Another survey conducted by Salo et al. [11] gathered data from 35 projects from 13 software organizations from eight different European countries. The survey was carried out using web-based questionnaire, and the goal was to find out the level of adoption and the usefulness of agile methods such as Scrum and XP. The study was carried out in 2006, so the data is rather old. This can be seen in the results: Scrum was used only in 27% of companies who participated in the study. When compared to the survey by VersionOne [16], the market share differs radically. Furthermore, Salo et al. [11] report that the most popularly used Scrum practice was product backlog. However, our results imply that the situation has changed and the most widely used Scrum practice is constituted by sprints, arguably reflecting the fashion the actual work is organized. Moreover, also other practices that are intimately associated with actual development activities, such as the daily Scrum, were widely used in the companies participating in the study.

A summarizing comparison of different benefits and issues of agile methods has been given by Petersen [9]. In addition to Petersen's work, the benefits and challenges of agile methods have also been studied by Tore Dybå and Torgeir Dingsøyr [5]. Their systematic article reviews several virtues and benefits of these methods as well as the associated problems and challenges. In general, these results fall along the lines of our survey as well.

VII. CONCLUSIONS

Due to the rapid pace of new discoveries in the field of software engineering, new practices are commonly taken use before conclusive evidence on their results is available. It is only after a while when the new practices mature and get adapted to the actual needs of the different companies, and the problematic elements of the practices can be identified. At this point, it is possible to gather empirical evidence on the use of the practices.

In this paper, we have introduced the results from a series of interviews in software industry, where the goal was to understand how widely Scrum is deployed, and what are the most notable deviations from Scrum recommendations, as well as the rationale behind these deviations. The list of anti-patterns derived in this work gives a good picture of Scrum practices that are in general found difficult to follow by companies. This list of anti-patterns can be exploited by practitioners taking Scrum into use in a company, to pay attention to seemingly attractive Scrum deviations that may nevertheless have unpleasant consequences. The list can also be used by researchers to understand the problematic aspects of Scrum in real life, and to provide guidance and practice refinements to avoid resorting to these anti-patterns.

Since in this paper we only addressed basic Scrum practices on the development side, the results are inconclusive when considering the integration of Scrum practices in the activities of companies. To gain better confidence on the results, the survey should be conducted on a wider range of companies. Furthermore, issues such as synchronizing the activities of individual Scrum teams into a larger system e.g. in the form of Scrum of Scrums [13] were not considered. Constructing a survey on such issues forms an interesting piece of future work.

ACKNOWLEDGMENTS

This work is partly funded by the Finnish Funding Agency for Technology and Innovation in the context of project Sulava. The authors wish to thank all the interviewed companies for their cooperation.

REFERENCES

1 Agile Alliance. Manifesto for Agile Software, Available at http://agilemanifesto.org (2001)

2 Brown W.J. et al.: Antipatterns - Refactoring Software, Architectures, and Projects in Crisis. Wiley 1998

3 Cockburn, Alistair. Agile Software Development, 1st edition, December 2001, pages 256, Addison-Wesley Professional, ISBN 0-201-69969-9.

4 Coplien, James O. and Bjørnvig, Gertrud. Lean Software Architecture for Agile Software Development. August 2010, pages 376 , Wiley, ISBN 978-0-470-68420-7.

5 Dybå, Tore and Dingsøyr, Torgeir, Empirical studies of agile software development: A systematic review. Information and Software Technology, Vol 50, Number 9-10, 833-859 (2008)

6 Easterbrook, S., Signer, J. Storey, M-A. Damian, D. "Selecting Empirical Methods for Software Engineering Research" In: Guide to Advanced Empirical Software Engineering, Shull, F., Singer, J., Sjöberg, D. (eds), 2008, ISBN 978-1-84800-043-8

7 Eloranta Veli-Pekka. Interview questions. Available at http://www.cs.tut.fi/~elorantv/interview_questions.html (2012)

8 Lindvall, M.; Muthig, Dirk; Dagnino, A.; Wallin, C.; Stupperich, M.; Kiefer, D.; May, J.; Kahkonen, T., "Agile software development in large organizations," *Computer* , vol.37, no.12, pp.26,34, Dec. 2004

9 Petersen, K. and Wohlin, C., A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. Journal of Systems and Software, Vol. 82, Number 9, 1479-1490 (2009)

10 Pikkarainen, M, Haikara, J., Salo, O. Abrahamsson, P. Still, J., The impact of agile practices on communication in software development. Journal of Empirical Software Engineering, Vol. 13, number 3, pp. 303-337 (2008)

11 Salo, Outi and Abrahamsson, Pekka, Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and Scrum. IET Software Vol. 2, Number 1, 58-64 (2008)

12 Schwaber, Ken, SCRUM development process. In: Proceedings of 10th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1995, p. 117-134.

13 Sutherland, Jeff, Agile Can Scale: Inventing and Reinventing Scrum in Five Companies. Cutter IT Journal, Vol. 14, Issue 12, (2001).

14 Sutherland, Jeff, ScrumBut Test aka The Nokia Test. Available at http://jeffsutherland.com/scrumbutttest.pdf (2010)

15 Sutherland, Jeff and Schwaber, Ken, The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game. Available at http://www.scrum.org/storage/scrumguides/ Scrum%20Guide%20-%202011.pdf (2011)

16 VersionOne 7th annual state of agile survey. Available at http://www.versionone.com/pdf /7th-Annual-State-of-Agile-Development-Survey.pdf (2012)

Artikel 10 bij leereenheid 4

Process Improvement for Small Organizations
Declan P. Kelly en Bill Culleton

IEEE Computer, volume 32, issue 10, October 1999

Process Improvement for Small Organizations

To introduce and sustain a software process improvement initiative, a smaller organization must minimize the limitations of its size and maximize the benefits inherent in its culture. The authors describe how this was done at Silicon & Software Systems.

Declan P. Kelly
Bill Culleton
Silicon & Software Systems

Software process improvement (SPI) has been a hot topic within the software industry for a number of years. Its high profile has been due in part to the introduction and industry acceptance of standard improvement models, most notably the Capability Maturity Model (CMM)[1] developed by the Software Engineering Institute at Carnegie Mellon University. Large organizations, such as Motorola's Government Electronics Division with 1,500 software engineers, have successfully implemented SPI initiatives and reaped significant benefits.[2] For smaller organizations, these benefits are no less significant, but smaller organizations often operate under different constraints.

For a small organization to introduce and sustain a process improvement initiative, it must minimize the limitations of its smaller size and maximize the benefits inherent in its culture. In this article, we describe an approach to SPI that has been used successfully in an organization of approximately 150 software engineers. Our approach involved as many software engineers as possible while avoiding disruption to ongoing projects.

SMALL-COMPANY ISSUES

The ratio of the effort required to implement a CMM-based SPI initiative in a company of 1,500 software engineers and a company of 150 software engineers will not be 10:1. In fact, for the core activities, excluding training and support, the effort required will not be significantly different. Clearly, the investment that a larger company can afford will usually be significantly greater than the investment that a smaller company can justify. Therefore, smaller organizations have an even more acute need to use SPI resources efficiently; if they don't, the cost of SPI can become prohibitively expensive.

Significant cultural differences exist between small organizations and larger ones. In smaller organizations, employees expect to be involved in all aspects of the software engineering process. Often, this practice is a result of the company's history. When a software company starts off with a handful of people, by necessity, everyone is involved in all aspects of software development. As these start-up companies grow, they often retain this culture of involvement. On the other hand, large organizations can leverage significant efficiency improvements by, for example, having a dedicated group choose computer-aided software engineering (CASE) tools or standards. Therefore, in a large organization, it is not unusual for decisions about the software process to be made outside the software engineering groups. In contrast, software engineers in smaller organizations expect to influence decisions that affect the way they work.

The culture of smaller organizations can often be characterized as creative, dynamic, and innovative. The success of these organizations is often due in no small part to the creativity and innovation of their employees. SPI is frequently viewed as the antithesis of these qualities, leading to bureaucracy that restricts the freedom of individuals. This aspect of small-company culture affects the SPI initiative in two ways. First, the SPI initiative should use the creativity of individuals within the organization to provide innovative solutions to SPI problems. Second, the result of the SPI initiative should not stifle creativity; it should focus creativity on project-specific problems, not standard process issues.

S3'S SPI PROJECT

Silicon & Software Systems (S3) has been providing design services in silicon, software, and hardware design since it was established in 1986. Within the software division, application areas include telecommunications, consumer electronics, Internet, and digital broadcasting.

Why the Capability Maturity Model?

We chose the CMM as the basis for our software process improvement because

- it provides a road map for process improvement through the five levels (see Figure A),
- it is an accepted industry standard and therefore allows easy comparison with other companies, and
- its key process area structure is flexible enough to tailor KPA implementation to suit the organization.

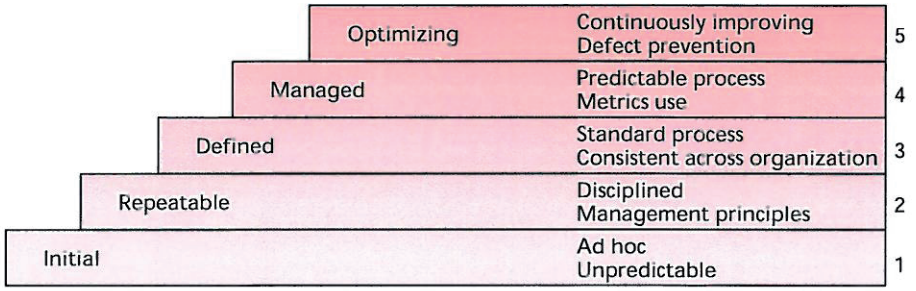


Figure A. The levels in the Capability Maturity Model.

The customer base is diverse and includes many major multinational companies. Because software projects in S3 cover a range of application areas and involve working with many different customers, the SPI process must be flexible enough to accommodate S3's diverse and constantly evolving projects. It would certainly be simpler to mandate a common programming language, design methodology, and set of CASE tools for all projects, but such an approach clearly is not an option for a company like S3. The diversity of S3's projects at first seems a complicating factor, but in practice it results in simplified procedures. This may seem surprising. However, applying quality procedures to such a diverse range of projects forces the procedures to focus on the essential details. The result is that the quality procedures mandate requirements essential to the quality of all projects, but they do not bound the freedom of projects by including arbitrary restrictions.

The starting point for our SPI initiative was an existing quality system—a set of quality procedures and tools to support their use. Throughout its history, S3 has used quality procedures, in many cases based on IEEE software engineering standards. The existing quality system had been certified ISO 9001-compliant.

Goals

The business objectives of our process improvement initiative were to reduce project lead time and improve the quality of the results, which in turn would increase customer satisfaction to the point that customers would keep using S3 as a preferred supplier. We had four main goals.

Maximize involvement, minimize disruption. In pursuing our SPI program, we wanted to involve as many of the software engineering staff as possible. There were two reasons for this. First, it would not be practical to have a dedicated group assigned full-time for the entire project. Second, given the organizational culture, it would not be acceptable to have solutions imposed by a small group.

All the people we wanted to involve were active on projects, which we did not want to disrupt. To deal with this, we broke the SPI work into small tasks, and we defined exactly what was required and how much effort it would take for each task. This approach allowed people to evaluate their project commitments and then commit to the SPI work only if it would not conflict with other project requirements. A single full-time coordinator was assigned to manage the project and coordinate the work of part-time participants.

We made sure that we involved people with varying degrees of experience from all sections of the software group. In particular, we tried to involve people who were skeptical about the need for a defined quality system and for a CMM-based SPI program. The skeptics generally think they know better and so object to what they perceive as restrictions on how they work. In practice, they often do have good ideas about ways of working; the trick is to channel their skepticism into improving the quality system. Involving them can often dissolve their skepticism. As they become immersed in the SPI initiative, they begin to see that it is not an attempt to impose restrictions but rather a way to improve the efficiency and effectiveness of the complete software group. Skeptics can also be some of the most influential individuals within the organization. Consequently, a benefit of converting them to SPI is that they influence others to become involved.

Stress quality, not CMM compliance. Although we were basing our initiative on the CMM, we did not focus exclusively on moving up the levels (see the "Why the Capability Maturity Model?" sidebar). Level 1 has no defined key process areas (KPAs) because it is the starting point; there *is* no defined process. In CMM terms, we were at Level 1 (although we had partially addressed all Level 2 KPAs), and we wanted to be formally assessed for Level 2. First and

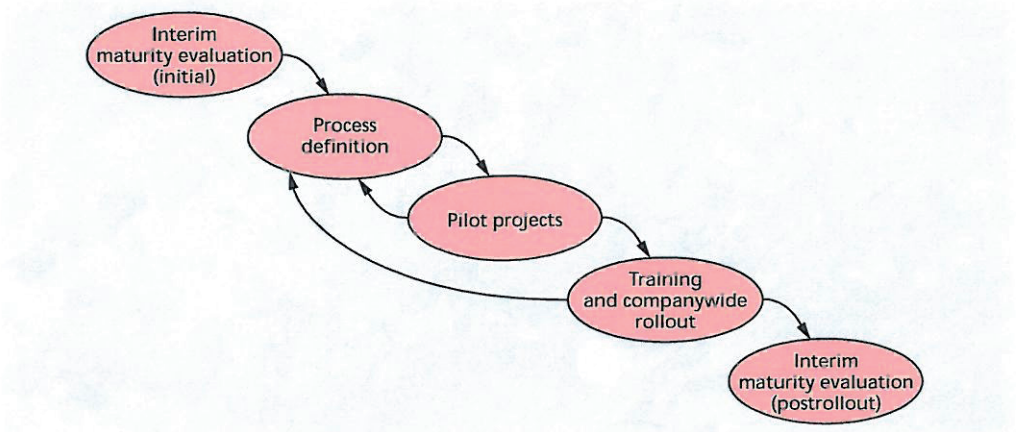


Figure 1. S3's software process improvement initiative. Improvement efforts began in September 1997 with an interim maturity evaluation to determine how S3 assessed relative to the CMM and its key process areas. The next step was to define a proposal for improving the weaker areas. Pilot projects helped validate the proposals. In June 1998, the team was ready to expand the process companywide and establish training. Shortly thereafter, we performed another interim maturity evaluation to gauge progress in addressing the key process areas.

foremost, however, we wanted an efficient and flexible quality system that would promote continuous process improvement.

Although achieving a particular CMM level will no doubt improve software quality,[1,3] there is always the danger of introducing a CMM-compliant process that does not fit your organization or that involves too much overhead. Thus, you have short-term process improvement, but it will be hard to sustain. If, however, the emphasis is on an efficient, flexible, quality system, the process improvement team is likely to keep working on the process definition until that efficiency and flexibility are achieved—even after a CMM-compliant process has been defined.

Also, CMM compliance should never be used to justify a process change. It is always tempting to counter objections to process changes by stating that the changes are a CMM requirement. It is more difficult, but ultimately more rewarding, to explain where the requirement comes from and then discuss the objections with a view to finding an acceptable solution.

Emphasize the advisory role. Most people view a quality system as a set of requirements to be enforced. We wanted the engineers to view it as both defining mandatory requirements and providing helpful advice. Our motto was, "Quality standards should seek to help the user do the job well, rather than just prevent the user from doing it badly."

To follow this motto, we structured the quality procedures in two layers. On the bottom is a set of mandatory requirements (which all projects follow). On the top is a set of optional guidelines, or best practices, which are based on the requirements. We deliberately put best practices in the guidelines layer because not all practices apply to all projects. The choice of which best practice to use is left to the project leader and the project team under the guidance of an independent quality assurance group. This is important because the final responsibility for a project's quality rests with that project's leader and team.

Promote efficiency. To make the quality system efficient, we wanted to be sure that the quality procedures were well written and clearly structured. People should be able to find the information they need without searching through multiple documents or following a chain of cross-references. We also wanted adequate tool support to make the quality system easy to use.

Interim maturity evaluation

Figure 1 shows the steps in our SPI initiative. The first step was a maturity evaluation. Before embarking on the SPI project, we wanted to evaluate our existing process against CMM KPAs. This would give us a baseline against which to measure our improvement. We based our evaluation on an Interim Maturity Evaluation Questionnaire—a set of questions, derived from the KPAs, about the way the organization worked. The evaluation consisted of a discussion based on the questionnaire and guided by an external consultant. The participants first individually scored each question and then as a group discussed questions they disagreed on. The discussion continued until participants agreed on the appropriate score for the organization for each question. We calculated an estimate of compliance for each KPA by averaging the scores for individual questions. In addition to providing us with an objective measure of our current status, the evaluation helped raise awareness of the issues we needed to address to achieve compliance at the next CMM level.

Where we were. In terms of CMM levels, S3 was a Level 1, but the typical description of a Level 1 organization as "unpredictable" (see Figure A) didn't apply. In our opinion, at that time, S3 most closely resembled the CMM Level 3 description: "standard process, consistent across organization." This situation is probably found in many Level 1 organizations: The company has not fully addressed all the Level 2

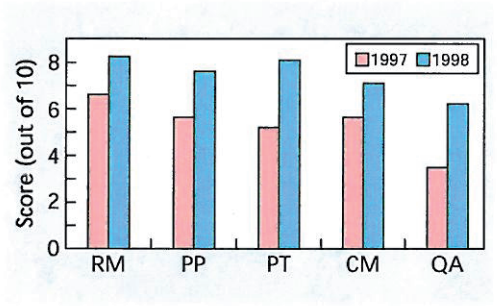


Figure 2. S3's compliance rating in the CMM Level 2 key process areas in September 1997, when the initiative began, and in June 1998, when the initiative was ready to go companywide. The Level 2 KPAs are requirements management (RM), software project planning (PP), software project tracking and oversight (PT), software configuration management (CM), and software quality assurance (QA). The software subcontract management KPA, also in Level 2, is not shown because S3 does not use subcontractors.

KPAs and so remains at Level 1, but it has partially addressed Level 2 and higher KPAs.

Figure 2 shows our compliance rating in Level 2 KPAs when we started our initiative. As the figure indicates, the existing quality system partially addressed all of these issues. It also covered some Level 3 KPAs, such as peer reviews. Many organizations starting a CMM-based SPI initiative will have some level of documented process. Starting an SPI initiative in a pure Level 1 company would be much more difficult.

Where we wanted to go. To achieve CMM Level 2, we needed to address requirements management, software project planning, software project tracking and oversight, software quality assurance, and software configuration management.

In addition to these CMM KPAs, we defined two non-CMM KPAs: completed work analysis and software metrics. CMM purists would object to including these with the CMM Level 2 KPAs because each represents a topic covered by a KPA at a higher CMM level. The CMM philosophy is that the KPAs at each level form a foundation for the process improvement at higher levels. Therefore, it isn't effective to implement higher level KPAs before the required foundation is in place. Although this may be true for a pure Level 1 organization, we started with a quality system that partially addressed all the relevant Level 2 KPAs and some Level 3 KPAs. We thus took a pragmatic approach and defined KPAs for issues we believed could offer significant benefits.

The completed work analysis KPA was a central part of our continuous process improvement plan. Its goal was to define a procedure for analyzing the work done on projects and for feeding back the experience so that it could be used in future projects. The scope was thus wider than process issues, encompassing tool use and communication and coordination on individual projects. We wanted to use the procedure from the completed work analysis to ensure that we learned not just from our mistakes but also from our successes. By analyzing the experience on all projects, we hoped to ensure that the lessons learned from individual projects were also learned by the complete organization. A group of 150 software engineers gains 150 person-years of experience every year. This represents a significant source of knowledge for the organization if it can be tapped.

The Level 2 software project tracking and oversight KPA covers some metrics, but we added a non-CMM KPA to define additional metrics. The goal was to define data to be collected by projects that would identify weaknesses in our existing process. Knowing the weaknesses would allow us to focus our process improvement effort on them. For example, we collected metrics on the defects introduced per project phase. This let us focus our improvements on the phases where the most serious defects are introduced. When defining the metrics to collect, we wanted to be clear about why we were collecting them, so for each metric, we also defined what we planned to conclude from the data.

Process definition

For the KPAs we defined ourselves (complete work analysis and software metrics), we had a clear understanding of what we wanted to achieve and the requirements. For the Level 2 KPAs, we chose to buy expertise so that we could get the project moving as quickly as possible.

KPA definition workshop. We organized a workshop and chose participants from across the software group so that all types of projects were represented. We did not assume that the participants had any knowledge of CMM, so the workshop started with a CMM introduction. The rest of the workshop focused on the Level 2 KPAs that were relevant for S3. The workshop's goal was to produce proposals for implementing these KPAs in the S3 environment. The consultant first described the requirements of two KPAs and gave some ideas about how to implement them. The participants then broke into two groups and worked out an implementation proposal. This approach was very effective. The groups were able to consider various options in the context of how people work in S3, and the consultant was there to answer any questions about CMM.

Once the two groups had agreed on a proposal, each group presented its proposal to the other, and the proposals were discussed further. At the end of the workshop, the participants had produced concrete proposals

Table 1. How the task force tailored an activity from the CMM Level 2 key process area (software project planning) to meet S3's needs.

Key process area	Description
Activity	The software engineering group participates on the project proposal team.
S3 context	The participants in the proposal are the sales team and software development team in conjunction with management. The sales department and senior management deal with the commercial and financial aspects.
Interpretation	Experienced software engineers prepare the technical aspects of the project proposal. These engineers normally form the core of the project team that will perform the work. At this stage, this team prepares an initial plan that documents the main aspects of the project. Once the customer and S3 senior management make a commitment in the form of a signed contract or letter of intent, the plan is further developed. Any further development or modifications at later stages must always involve the project team as well as any organizations outside the project that are affected by it.

for implementing two CMM KPAs. The success of the workshop was due in large extent to the consultant, who not only understood the CMM theory but also had experience with its practical application. This was a great help to us as we designed an implementation of the KPAs to suit our working environment.

One striking outcome of the workshop was that all the participants, without exception, were enthusiastic about the SPI project and were highly motivated to contribute further. They all agreed to meet again the following week to work on the remaining KPAs. This follow-on workshop also resulted in implementation proposals.

Task forces. To involve as many people as possible in the SPI project, we used task forces. In each task force, one person was responsible for implementing the changes, and approximately five people reviewed and guided the implementation. Each CMM Level 2 and internally defined KPA was assigned to a task force. This ensured that a large proportion of the organization was directly involved in the SPI project. Indeed, after this step in the initiative, approximately 45 percent of the software group had been directly involved.

Each task force followed a standard approach to KPA implementation. First, they wrote a proposal describing how the KPA would be implemented. For CMM KPAs, they produced a detailed proposal document that was based on the workshop results. For the non-CMM KPAs, they wrote a proposal document. In each case, the task force reviewed and discussed the KPA proposal and agreed on how to implement the KPA. The next step was to implement the agreed-on changes. This involved updating existing quality procedures, writing new procedures, and, in some cases, providing tool support. The task force also reviewed any new or updated quality procedures. Table 1 shows how the task force proposed to implement an activity from the software project planning KPA.

Pilot projects

Once the task force approved the changes for a KPA, we were ready to introduce them on real projects. For KPAs that had significant changes, we chose to test the changes on a pilot project. This allowed us to monitor the effects of the changes very closely and take corrective action if problems arose with the KPA requirements.

We tested our changes to the software project planning KPA on an Internet application project that had a short life cycle. The project let us not only verify the completeness of the process, but also test the downward scalability of the KPA definition to a small project. The project team implemented all the improvements and tracked their effects. From the customer and the S3 management point of view, the documented project plan provided excellent visibility and greatly aided tracking. The project team was more confident in their plan and, because they spent less time replanning, as had been typical of previous projects, they had more time to concentrate on technical issues and product quality. Also, increasing the level of detail (relative to previous plans) produced a more complete schedule, thus reducing the number of tasks that might otherwise have been forgotten.

In general, the pilot project test of the project planning KPA definition went very smoothly, and the KPA definition did not change significantly.

In contrast, the pilot project to test the quality assurance KPA definition did encounter problems. After less than a month, it became apparent that the KPA definition was not practical, although it had been set and reviewed by many people and seemed to be a reasonable approach. The amount of overhead to both the project and supporting infrastructure was not acceptable, and the return on invested time was questionable. When we analyzed the KPA definition, we found that we had spent too little time tailoring the KPA to S3's environment, and we had not sufficiently defined

We tried to get as much feedback as possible from the software engineers who would use the updated quality system.

the supporting infrastructure. The project immediately stopped using this KPA definition, and the original task force, in conjunction with the pilot project team, redesigned it. We put the new version and an improved infrastructure in place only after the project was completed. The updated KPA definition has since been tested and is proving effective.

Training and rollout

Before introducing the updated quality system throughout the software group, we arranged training on the new system for all software engineers and relevant managers. To stimulate discussion, the training was conducted in groups of 12. The training course for each group lasted a full day. During the course, we tried to get as much feedback as possible from the software engineers who would use the updated quality system. Also, we tried to instill an understanding of the context and purpose of the changes so that the engineers would not only understand what was required of them but also why. The training courses included a walk-through of some of the updated quality procedures, which generated much useful discussion and many suggestions for improvements.

The training course initiated the rollout of the new processes to all projects. Process mentors were assigned to each project to support the introduction of the new processes. For existing projects, the process mentor and project leaders decided case by case which parts of the new processes to adopt and at what point to introduce the changes.

Infrastructure. To make the updated quality system easy to use, we developed a supporting infrastructure and used the companywide intranet to support it. We provided an easy-to-use Web interface to the quality documents and tools to support specific changes that resulted from KPA implementation. Among these was a lessons-learned database that provided easy access to suggestions arising from the completed work analysis performed on projects. A software process group also reviews these lessons and looks for ways to provide feedback into the quality system.

Version control. Our goal to start a continuous improvement process conflicted with the need for stability within the quality system. If the quality procedures are constantly changing, the multitude of versions will cause confusion. To avoid this, we released modified documents only when we released a new quality system version, which was at most every three months. With each quality system version release, we provided a full description of all changes. This gave projects visibility into any changes that had been introduced. When a project starts, it uses the latest version of the quality system. A project might later adopt a newer version of the quality system, depending on how significantly it has changed. Typically, projects will change versions only at the start of a major new phase.

RESULTS

In evaluating an SPI project, you cannot consider only objective measurements because the software process is ultimately about helping people do their jobs. Trying to measure process improvement while ignoring the people involved provides a very one-sided view. If the people who use the quality system do not feel that the SPI is improving things, there is a problem.

Measurements

At the start of our SPI project, we conducted an interim maturity evaluation to assess our status with respect to CMM KPAs. After we introduced the new quality procedures and tool support, we again performed an interim maturity evaluation. Figure 2 shows the results of this evaluation.

When we started our SPI initiative, we were reasonably strong in all areas except quality assurance. The strength in the other KPAs was due to our ISO 9001-compliant quality system, which in our opinion was already a very strong foundation. Figure 2 shows real improvements in all KPAs. The most striking improvement is in quality assurance—understandable because this was our weakest area initially and hence where we could gain the most. The large improvement is due to CMM's emphasis on a supportive role, for example, support at the planning stage, in contrast to ISO 9001's audit-based approach. The figure also shows that we can benefit even more, especially in quality assurance. We decided that before seeking a formal CMM assessment, we should score at least nine for each KPA in the interim maturity evaluation. Moreover, given that our primary goal was to achieve real improvements, with CMM certification as a secondary goal, we also decided to look at further improvement benefits before seeking the assessment.

Since the first rollout in June 1998, all projects have moved to the new process definitions, with the help of the training course and mentoring by process experts. The feedback from customers has been very positive. Before the project, most of our customers required that we use their process; now, after one and a half years, most customers are happy to see projects use the S3 process. An excellent example is S3's recent recognition as a qualified software supplier to a CMM Level 4 company. The positive feedback from customers proves that the SPI program is contributing to S3's business goals.

Subjective evaluation

The subjective evaluation of our SPI project is also very positive. Software engineers feel greater owner-

ship of the quality system and are more inclined to suggest improvements through a structured change proposal mechanism, as evidenced by the increased number of change proposals. They also greatly appreciated our addition of a strong support infrastructure. Managers feel that the SPI initiative has given them greater visibility, though not yet complete, into projects and their progress.

The effort expended to date, excluding the project leader, is 180 person-days on process redefinition, 70 person-days on training, and 20 person-days on evaluations. Two new procedures were introduced, one on requirements management and the other on risk management (as part of project planning). We have revised six existing procedures, one of which is the project planning documentation procedure.

Our approach to process improvement has proved very effective. There has been a noticeable shift in attitudes toward the quality system. Where once it was seen as something static to be tolerated, it is now seen as a living thing that is constantly evolving. Thus, although not everyone agrees with all aspects of the quality system, people are far more willing than in the past to make constructive suggestions.

The SPI approach described in this article was targeted for S3's software division. A process improvement initiative that follows a similar approach is now being introduced across S3. ❖

Acknowledgments

We thank Dave Murrells of Silicon & Software Systems for reviewing an earlier version of this article and providing valuable comments.

References

1. K.M. Dymond, *A Guide to the CMM: Understanding the Capability Maturity Model for Software*, Process Transition Int'l, Annapolis, Md., 1995.
2. M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software*, Sept./Oct. 1997, pp. 75-81.
3. J. Herbsleb et al., "Software Quality and the Capability Maturity Model," *Comm. ACM*, June 1997, pp. 30-40.

Declan P. Kelly is a research scientist at Philips Research Laboratories in Eindhoven, The Netherlands. Before joining Philips Research, he worked in the software division at Silicon & Software Systems as software process improvement coordinator, among other roles. His research interests include digital video processing, storage applications, software engineering, formal methods, and software process improvement. He received an MSc in computer science from Trinity College, Dublin, and a joint honors BSc in mathematics and computer science from University College, Dublin. He is a member of the IEEE Computer Society and the ACM.

Bill Culleton is process and quality manager at Silicon & Software Systems, Dublin, where his responsibilities include managing a CMM-based process improvement project in the software division as well as project management mentoring and training. He also supports process improvement in S3's other divisions. He received an MSc in microelectronics design and a BA BAI in telecommunications engineering from Trinity College, Dublin. He is a member of the Institute of Engineers of Ireland. Contact Culleton at bill_c@s3group.com.

Artikel 11 bij leereenheid 4

Successful Process Implementation
Anna Börjesson en Lars Mathiassen

IEEE Software, volume 21, issue 4, July 2004

Successful Process Implementation

Anna Börjesson, *Ericsson*

Lars Mathiassen, *Georgia State University*

How do you measure success in software process improvement? The answer is perhaps not as straightforward as it seems. Success is traditionally measured as the difference in quality and productivity between the old and new engineering practices.[1] However, this measure requires systematic benchmarking and data collection over long periods of time, and few software organizations can meet this demand. So, how can we more realistically measure SPI and practically guide our SPI efforts toward success?

We propose measuring SPI success through *implementation success*—the extent to which initiatives lead to actual changes in software engineering practice. First, without implementation success, SPI success is impossible. Second, only when implementation succeeds can we see how SPI initiatives affect software practices. Third, implementation success is easy to assess. Finally, focusing on implementation success is a pragmatic way to steer SPI initiatives toward success.

We studied the approach and outcome of 18 different SPI initiatives conducted over a five-year period at the telecom company Ericsson AB, based in Gothenburg, Sweden. Doing so gave us insight into how SPI initiatives can best

- Ensure stakeholder commitment
- Support organizational learning
- Distribute resources over different activities
- Manage customer relations

Traditional approaches to measuring software process improvement are typically lengthy, data intensive, and cost prohibitive. A simple indicator, the extent to which engineering practices change, can provide enough information to guide initiatives toward success.

The Ericsson experience

Ericsson has provided packet-data solutions for the international market for more than 20 years and is among the world leaders in this area. Between 1995 and 2001, the company grew from 150 to 900 employees. During this period, SPI played a key role in improving software productivity and quality, and the company conducted many SPI initiatives with varying degrees of success. The initiatives were executed in the same organizational context and, in most cases, involved the same SPI people. The initiatives also followed the same IDEAL approach[2] to SPI (see "The IDEAL Model" sidebar): After *initiation*, the initiatives went through one or more cycles of *diagnosing* problems, *establishing* focused improvements, *acting* to improve, and *learning* from results.

However, the SPI initiatives had important differences (see Table 1). They focused on different improvement areas, had different vol-

The IDEAL Model

The IDEAL (*I*nitiating, *D*iagnosing, *E*stablishing, *A*cting, and *L*earning) approach,[1] developed in 1996 by the Carnegie Mellon University Software Engineering Institute (www.sei.cmu.edu/ideal), presents a five-phase, cyclic approach to software process improvement.

The initiating phase (see Figure A) is where you establish the initial improvement infrastructure, define the initial roles and responsibilities for the infrastructure, and assign initial resources. You create an SPI plan to guide the organization through the completion of the initiating, diagnosing, and establishing phases. Also, you obtain approval for the SPI initiative along with a commitment of future resources for the tasks ahead.

The diagnosing phase lays the groundwork for the later phases. The SPI action plan is initiated in accordance with the organization's vision, strategic business plan, lessons learned from past improvement efforts, key business issues the organization faces, and long-range goals. You perform appraisal activities to establish a baseline of the organization's current software operation. You reconcile the results and recommendations from the appraisals with existing and planned improvement efforts for inclusion into the SPI action plan.

During the establishing phase, you prioritize the issues that the organization has decided to address with its improvement activities and develop strategies for pursuing the solutions. You detail and complete the SPI action plan on the basis of the adopted strategy and the decisions made. You design focused projects to address each prioritized improvement area.

In the acting phase, you create, pilot, and deploy throughout the organization solutions to address the areas for improvement discovered during the diagnosing phase. You develop plans to execute pilots to test and evaluate the new or improved processes. After piloting these processes and determining their readiness for organization-wide adoption, deployment, and institutionalization, you develop and execute plans to accomplish the rollout.

The learning phase (called the "leveraging" phase when IDEAL debuted) aims to make the next pass through the IDEAL model more effective. By this time, you've developed solutions, learned lessons, and collected metrics on performance and goal achievement. These artifacts are added to the process database as a source of information for personnel involved in the next pass through the model. Also, on the basis of this information, you can evaluate the strategy, methods, and infrastructure used in the SPI program. By doing this, you can correct or adjust the strategy, methods, or infrastructure prior to executing the IDEAL model's next cycle.

Depending on the resources organizations commit to their SPI program, they can pursue IDEAL activities in parallel, and some parts of the organization can pursue activities in one phase of the model while others pursue activities in a different phase.

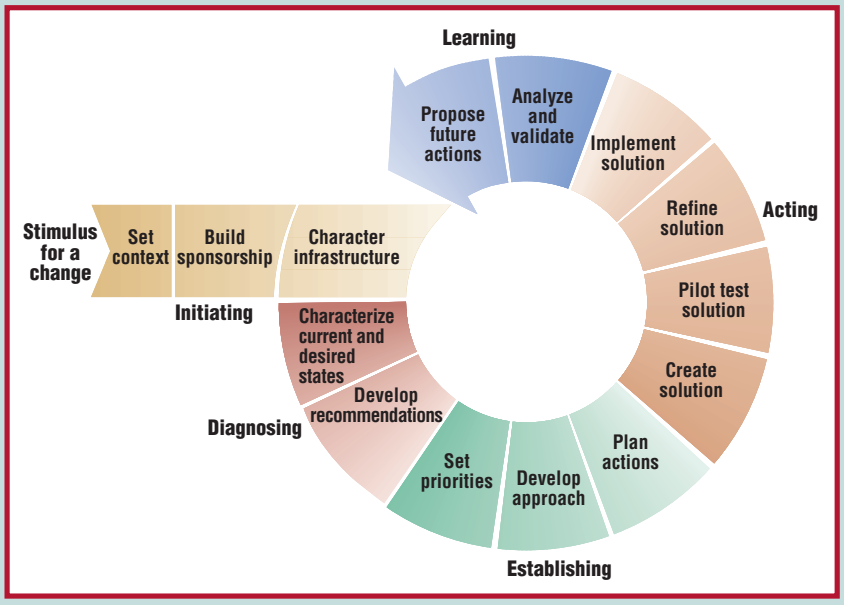


Figure A. The IDEAL model presents a five-phase approach to software process improvement. (Special permission to reproduce "Ideal Model Graphic," © 2003 by Carnegie Mellon University, is granted by the Software Engineering Institute.)

ume, and targeted different parts of the organization. In addition, the initiatives adopted assorted improvement tactics and went through a varying number of IDEAL model cycles. Also, while commitment is generally recognized as a key factor in SPI success,[1,2] different social forces drove each initiative.[3] The *process push* depends on the competence, commitment, and active participation of *process engineers* in developing and implementing new engineering practices. The *practice pull* instead depends on the competence, commitment, and active participation of *software practitioners* in developing and adopting new practices. These differences in improvement tactics (that is, the number of iterations, degree of process push, and degree of practice pull) resulted in varying levels of implementation success.

SPI participants collected data both during and after the initiatives using time-reporting systems, project specifications, final reports, and interviews with process engineers and

Table 1

Process implementation data on Ericsson's software process improvement projects

SPI initiative	Improvement area	Volume	Target	Ideal iterations	Process push
1	Configuration management	10 weeks 300 person-hours 4 participants	Several units	1 full cycle	Weak. Process engineers focused on designing a generic process, with no commitment to or plan for actually deploying the process.
2	Design information	21 weeks 400 person-hours 6 participants	Several units	1 full cycle; stopped during establishing in the second cycle	Weak. Process engineers focused on designing a generic process, with no commitment to or plan for actually deploying the process.
3	Estimation and planning	14 weeks 600 person-hours 11 participants	Several units	1 full cycle; stopped during establishing in the second cycle	Weak. Process engineers focused on designing a generic process. Time for mentoring and process deployment was limited.
4	Historical data	16 weeks 200 person-hours 4 participants	Several units	1 full cycle	Weak. Process engineers focused on identifying historical data, little of which had been recorded. SPI participants planned very few activities to communicate the results.
5	Introductory training	14 weeks 620 person-hours 11 participants	Several units	1 full cycle	Weak. Process engineers were dedicated to defining and describing the process. No plans for how to deploy the process and support its actual use.
6	Module tests	12 weeks 400 person-hours 10 participants	Several units	1 full cycle; stopped during establishing in the second cycle	Weak. Process engineers focused on designing a generic process. Too little time was planned to help the project actually use the result.
7	Project tracking	9 weeks 300 person-hours 7 participants	Several units	1 full cycle; stopped during establishing in the second cycle	Weak. Process engineers were given time only to define a process, not to implement it in projects.
8	Resource handling	4 weeks 250 person-hours 8 participants	Several units	1 full cycle	Weak. Process engineers focused on solving the problem through a well-defined process.
9	Requirements management	10 weeks 200 person-hours 5 participants	Several units	1 full cycle	Weak. Participants spent most of the time defining requirements management; they planned little time to help implement the results.
10	Requirements management implementation	12 weeks 330 person-hours 7 participants	Project	1 full cycle; stopped during establishing in the second cycle	Strong. Management made time for the process engineers to help implement the project results.
11	Subcontract management	18 weeks 650 person-hours 9 participants	Several units	1 full cycle	Weak. Process engineers were strongly committed to solving the problems, but the resulting process wasn't grounded in current practices. No time was planned for activities to make change happen.
12	Requirements management	30 weeks 1,200 person-hours 3 participants	Project	4 full cycles; stopped during diagnosing in the fifth cycle	Strong. Management made time for process engineers to mentor and support the project in action. The SPI initiative was dedicated to solving the problems for one project.
13	Analysis and design	30 weeks 1,000 person-hours 4 participants	Unit	5 full cycles; stopped during establishing in the sixth cycle	Strong. Process engineers planned the deployment activities and made time available for mentoring and support.
14	Implementation	30 weeks 1,000 person-hours 4 participants	Project	4 full cycles	Strong. Management gave process engineers time to participate in the project to support implementing the results.
15	Test	30 weeks 1,300 person-hours 2 participants	Unit	4 full cycles	Strong. Process engineers participated in software tests and came to understand the tester's specific needs.
16	Configuration management	30 weeks 1,650 person-hours 6 participants	Unit	4 full cycles; stopped during diagnosing in the fifth cycle	Strong. Management dedicated sufficient resources to daily mentoring and support to make the change happen.
17	Project management	10 weeks 150 person-hours 2 participants	Unit	3 full cycles	Strong. Some process engineers were also customers who would be using the results. The SPI commitment to the initiative was very high.
18	Process development map	30 weeks 200 person-hours 2 participants	Unit	4 full cycles	Strong. Process engineers saw the need for a well-defined process development map to help communicate and deploy all SPI work.

Practice pull	Implementation success
Weak. Projects focused mainly on making generic process descriptions. Practitioners were eager to solve problems on a general level but had little commitment to using the results.	Low. Potential users considered results hard to use. SPI participants used part of the result indirectly, when they applied the knowledge gained to other projects.
Weak. Projects focused mainly on making generic process descriptions. Practitioners were eager to solve problems on a general level but had little commitment to using the results.	Medium/low. The intention was to provide a design framework. The results were mainly implemented in one project that one process engineer worked on.
Medium. Projects were dedicated to solving generic process problems, but only one project manager was interested in actually testing the results.	Low. The results were tested in one project, but the project ran into difficulties. Support was weak, and no one helped the project; resources from the SPI initiative were no longer available to make necessary changes.
Weak. Managers were eager to find out about historical data. When they found very little, their commitment to change dropped dramatically.	Low. The purpose was to build a database of old data and take action from there. Participants found no interesting data and thus made no changes.
Strong. The result was focused on supporting managers who were asking for help and interested in applying the results.	Medium. An estimated 50 percent of managers used the process. Some didn't know about it and weren't given the opportunity to learn about it. The SPI initiative gave managers supporting guidelines. More assistance was sometimes needed.
Weak. Many practitioners believed in performing systematic module tests, but no one was committed or given the time to implement the results.	Medium/low. The process was used only when process engineers were members of a project or when section managers strongly believed in systematic module tests.
Weak. Most project managers believed good follow-up on a project was necessary, but only one was committed and willing to try out the results in practice. Everyone else waited to see if someone else benefited from the process.	Medium/low. Only one project, which was supported by the SPI initiative's driver, used the process. That project team was content with the outcome.
Medium. The managers believed in supporting resource handling, and most of them were willing to use the result.	Medium. An estimated 75 percent of managers used the new process. Those not using the results either didn't get the help they needed or didn't believe in the approach.
Weak. Everyone knew that managing requirements was important, but no one was committed to acting on the results in his own project.	Low. The results were hard to use. They were used mainly as a framework by the members of the SPI initiative.
Weak. The targeted project was interested but not committed to spending the time required to make the change happen.	Medium. Initiative 9's low impact spurred this initiative. This initiative focused on one project, and all results were designed to suit its needs.
Weak. Managers were strongly committed to creating better routines for subcontract management, but no time was planned for the projects to actually implement the new process.	Low. The results needed further adaptation to be useful for different projects, but no one tried to tailor them. Project engineers used some results indirectly in other projects.
Strong. A few highly respected practitioners were strongly committed to the initiative. They helped ensure the results would actually be used.	High. The process was adapted to a specific project but needed further adaptation to be useful. Process engineers and practitioners solved these problems jointly and made the change happen.
Strong. A few well-respected practitioners were convinced (after a few difficult weeks) of the need for improved practices and committed to making them happen.	Medium/high. This is a complex area and required several iterations of experimenting with processes before the result was satisfactory.
Strong. The practitioners were receptive to adopting a stronger focus on implementation, and they participated actively in the change process.	High. Two slightly different adaptations were made to fit the needs of different products developed on different sites. Collaboration between process engineers and software practitioners made the change happen.
Strong. The practitioners came to understand that the SPI initiative was responsive to their specific needs, and they became committed to using the results.	High. The result was adapted to the specific needs of a specific project. Process engineers and software practitioners solved difficulties together.
Strong. The configuration managers were highly involved in both defining and deploying the results.	High. Each specific situation has hundreds of possible solutions. Choosing one and focusing on making that happen required extra attention. The software practitioners' dedication played a key role.
Strong. The project managers wanted to implement their own ideas and took the time to do it.	High. The results were actually used, and the project managers' dedication to SPI work within project management continued.
Strong. All software practitioners needed a process description that would provide templates, guidelines, and other relevant information.	High. The development-process map's use is measured in both "hits" to a site on the process web and subjective opinions of need. All measurements are very positive.

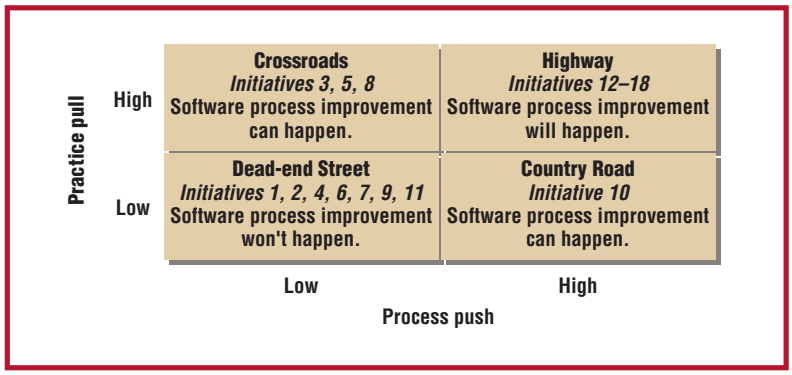


Figure 1. Four roads to process implementation.

software practitioners. One of us was directly involved in and responsible for all of these initiatives; the other had conducted SPI research in many other organizations. Analyzing the data revealed how important it is to actively manage commitment, learning, resource distribution, and customer relations if SPI efforts are to succeed.

Managing commitment

Categorizing Ericsson's SPI initiatives on the basis of the degree of process push and practice pull, we arrive at four different roads to process implementation (see Figure 1).

The Dead-end Street initiatives focused on the process itself—specifically, on process definition and specification. The initiatives targeted several Ericsson units, but the new process was often hard to apply because process engineers had to make many compromises to meet everyone's needs. The process push toward implementation was low because the process engineers focused on defining a generally applicable process. Few resources and incentives existed for the process engineers to become engaged in the involved units' different engineering cultures. The practice pull was also low because it was difficult to engage software practitioners from different units in one common initiative. Needs and backgrounds differed across the units, and they had no tradition for communication and collaboration across units. The Dead-end Street initiatives never amounted to much. No one was seriously committed to implement the processes, so the organization saw little overall benefit.

The Country Road initiative targeted a specific project's engineering practices. The process engineers worked in the project and had time to help implement the results. The process engineers, therefore, understood the particular culture and practices in the project, and they were strongly committed to support and change practices. They focused on requirement issues and created a process that fit the project's particular needs. However, the practitioner commitment was weak. The software practitioners weren't motivated to change requirements practices. They didn't understand why they had to be involved and didn't allocate time to work with process implementation. The process push was high, but the practice pull was low. A typical Country Road initiative can happen, but it's slow going and likely to fall short of changing engineering practices.

The Crossroads initiatives targeted several company-level units with similar requirements. Practices and needs across the units had little variation, so process compromises were unnecessary. The practice pull was high: Software practitioners understood the need for new processes and were committed to using them. But the process engineers failed to allocate sufficient time and resources to implement the process, so the process push was low. A typical Crossroads initiative can happen, and at Ericsson, the initiatives might succeed. However, the software practitioners might yet face barriers to effective implementation,[4] and process engineers are no longer available to guide them and facilitate the change process.

The Highway initiatives targeted practices in a single unit or project. The main focus was on solving specific problems identified by software practitioners, and they required few compromises. Both process push and practice pull were high. The process engineers were committed and allocated time to process implementation, and the software practitioners understood why they needed the new approaches and appreciated the SPI initiative. The process engineers and the software practitioners worked closely together and communicated intensively about needs, problems, and progress. A typical Highway initiative will happen, and engineers will implement and use the results. Organizations directly benefit from such initiatives.

Managing learning

Successfully changing software practices requires learning, and helping organizations learn is by no means easy.[5,6] Iterations support learn-

ing by letting you correct failures and modify processes based on practical experience. Although CMM founder Watts Humphrey doesn't use the word *iteration*, he does discuss the importance of performing SPI work in steps and repeatable sequences.[1] Several SPI approaches emphasize iterative development, including the IDEAL model[2] and methods that implement plan-do-act-check cycles.[7]

We examined how Ericsson's SPI initiatives used iterations and feedback from practice to support learning. Figure 2 illustrates the relation between the initiatives' implementation success and the number of iterations they executed. As the figure shows, the number of iterations significantly affected SPI implementation success: As the number of iterations increased, so too did implementation success.

However, because factors apart from iterations affect implementation success, this pattern has exceptions. Initiatives 5 and 8, for example, executed a single iteration but still achieved moderate implementation success. One possible explanation is that both initiatives had high practice pull. Also, Initiative 13 executed two iterations in the last phase alone and still failed to achieve high implementation success, while Initiative 17 executed only three iterations total and was highly successful. Compared to the other initiatives, however, Initiative 13 was by far the most complex and difficult and consumed the most time and expertise. In contrast, Initiative 17 had an experienced project manager who exercised considerable practice pull. So, projects can succeed with few iterations. In general, however, more iterations support more learning and better facilitate change.

To further illustrate the importance of iterations, we mapped each initiative to Gerald Weinberg's four phases of successful change (see Figure 3).[8] As the figure shows, initial attempts to introduce a new process lead to chaos. The process then becomes integrated as engineers attempt to practice it, eventually giving rise to a new and stable status quo of engineering practices. However, each phase has barriers that make implementation difficult, and these barriers can cause the process to regress to previous phases.

As Figure 3 illustrates, an initiative typically took one full iteration to pass one phase in Weinberg's change model. Most SPI initiatives must pass the chaos phase and enter the integration and practice phase to successfully implement a new or modified process. SPI initiatives that execute only a single iteration will therefore seldom enter the phase of using the new process as an integral part of engineering practices. Thus, the process simply won't be adopted as intended.

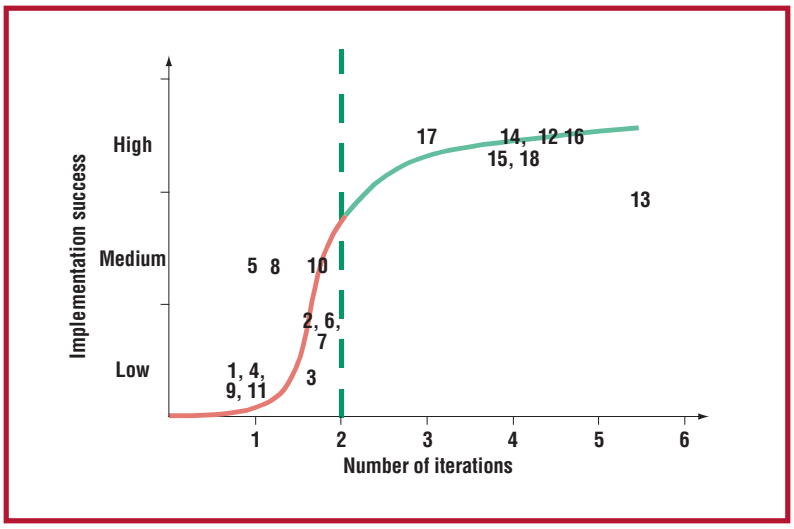


Figure 2. Implementation success and number of iterations (1 through 18). More iterations meant more implementation success.

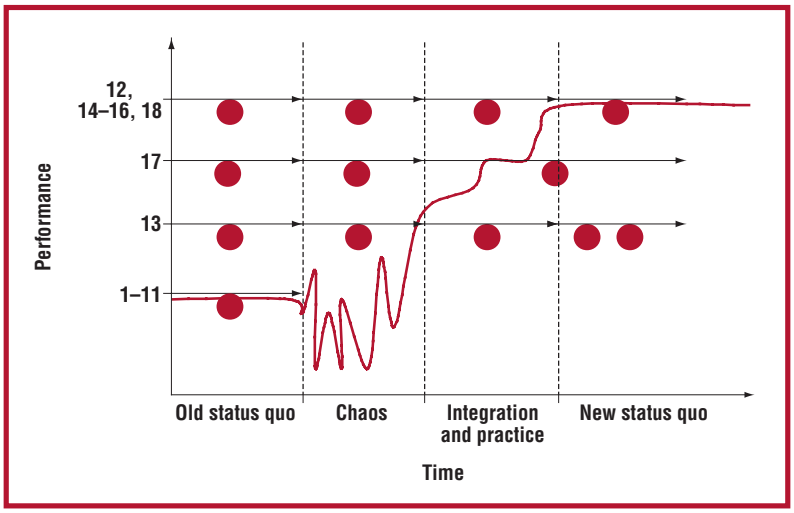


Figure 3. Performance and distribution of iterations (numbered 1 through 18). The jagged line illustrates Weinberg's optimum learning curve. The circles denote the number and distribution of iterations over phases.

Managing resources

As Humphrey says, "SPI requires investment—it takes planning, dedicated people, management time, and capital investment," and, for an organization to improve, "someone must work on it."[1] To drive SPI work toward success, organizations must commit and manage their resources accordingly, and they

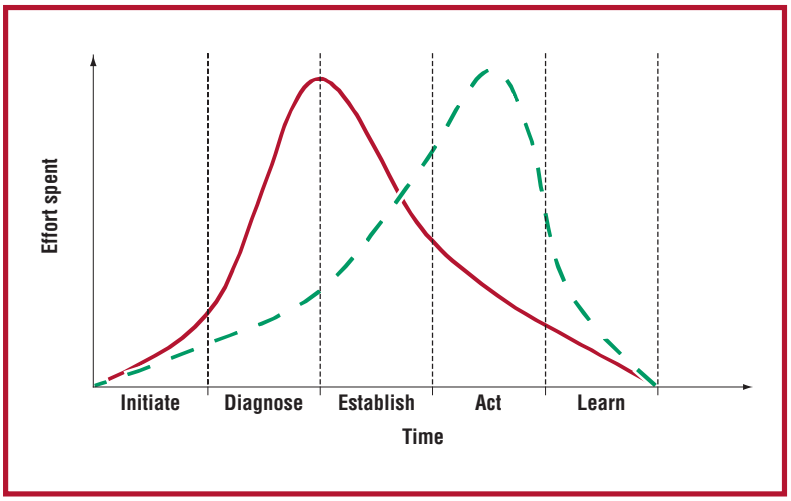


Figure 4. Effort spent over IDEAL phases. Initiatives that overinvested in early phases, represented by Initiative I (the red curve), had low implementation success. Initiatives that invested most resources in later phases, represented by Initiative I2 (the green curve), had high implementation success.

must constantly motivate people to participate and contribute. Any significant change can create substantial fear and uncertainty throughout the organization. It's essential that process engineers possess change-management skills.[8] Skillful change agents can work closely with software practitioners to solve potential difficulties that arise regarding both the new process and the change process itself.

Ericsson's process engineers managed resources quite differently in the SPI initiatives. Figure 4 shows how the process engineers distributed their time over the IDEAL model's phases[2] in two ways. Initiative 1 is representative of the pattern in initiatives 1 through 11. It had low implementation success, and engineers invested most of their effort into the model's early phases (the red curve). Initiative 12 is representative of initiatives 12 through 18. It had high implementation success, and the process engineers invested most of their effort in the model's later phases (the green curve). As this comparison indicates, spending considerable effort in the model's later phases might be more successful than putting most effort into early phases. Most of Initiative 1's process-engineering resources were used to analyze, design, and describe the new process. This meant few resources were available during the difficult implementation phases, in which software practitioners can experience significant fear and uncertainty. Because Initiative 12 invested considerable resources in these later phases, the process engineers could better manage and address change issues as they emerged.

Managing customer relations

To achieve SPI success, process engineers must have a positive, collaborative relationship with their customers, the software practitioners.[9] Positive customer relations will help process engineers better understand engineering practices and problems and thus develop more useful processes. A good relationship also helps software practitioners overcome their resistance to change: When the status quo is challenged, they often become defensive and alienated.[8]

For managing customer relations, a dedicated SPI approach seems more likely to succeed. As Table 1 shows, Initiatives 1 through 11 generally supported several units and thus used a relatively generic SPI approach that achieved moderate success at best. Initiatives 12 through 18 supported only one project or unit and used a more dedicated approach with greater overall success. Of course, other factors influence SPI outcome, and rules always have exceptions. Nonetheless, generic approaches make it difficult for process engineers to build and maintain good customer relations—there are too many different relationships, needs, and requirements. In dedicated approaches, process engineers and software practitioners can work closely together and focus on the project's specific challenges. Also, in generic approaches, it's difficult to handle the many personalities and requirements when relationships with the software practitioners become fragile. In such situations, it's much easier for process engineers to simply focus on the process design and description and forget about troublesome implementation issues.

Lessons learned

Our experiences with the Ericsson initiatives offered several lessons about how organizations can more successfully manage SPI.

Focus on implementation

When it comes to software, we already know that test-and-repair strategies fail to deliver quality. To achieve quality, software projects must consider it and plan for it from the

start. Similarly, in an SPI initiative, you should consider implementation issues early on. Such considerations should include[10]

- Critically evaluating the new process from an easy-to-adopt standpoint
- Examining the roles that stakeholders must play during implementation
- Choosing an implementation strategy that suits the initiative
- Assessing and resolving implementation risks
- Outlining an initial plan for implementation

Such early focus on implementation can help you design processes that software practitioners are more likely to adopt and thereby reduce the risk of failure.

Take the Highway

To succeed with SPI, you need a serious commitment from key stakeholders.[1,7] To develop this commitment, you must ensure that process engineers have sufficient change-management skills and resources to create high process push. This in turn facilitates the active involvement of software practitioners to create high practice pull. Also, it's important to focus on the cultural environment of the change process.[11] Are the involved actors motivated? Are senior and middle managers involved? Are the recognition and reward systems appropriate? And, is communication between the involved actors supportive? The Highway is the best route to SPI implementation success. The Country Road and Crossroads are risky and require additional attention and resources to create complementary pull and push, respectively. As its name implies, the Dead-End Street leads nowhere.

Iterate, iterate, iterate

SPI initiatives that execute only a single iteration never enter the phase in which the new process is exposed to practice. In such initiatives, the process engineers get no feedback on whether the process is useful, and the process won't likely be used. SPI initiatives that execute several iterations are more likely to enter and pass through the phase in which practitioners resist change; when the defined process meets actual practice, process engineers can learn and react. The result is a process that practitioners will likely use. If you plan for several iterations, you're more likely to overcome resistance to change and facilitate learning.

Expect chaos

At Ericsson, process engineers initially perceived some initiatives, such as Initiative 1, as successful because the operations went smoothly. They viewed other initiatives, such as 12, as problematic because they caused debate, anxiety, and active resistance among software practitioners. Paradoxically, Initiative 1's implementation ultimately met with low success, while Initiative 12 was highly successful. The explanation is simple: If you emphasize process analysis and design, you won't experience the tension that arises between your design and current engineering practices. If, on the contrary, you engage in process implementation activities, you'll be confronted with misfits and psychological reactions. You should therefore expect a certain level of chaos in SPI initiatives. Chaos is often a sign that the implementation process is on its way and that you're about to receive valuable information that will help you succeed.

Chaos is often a sign that the implementation process is on its way and that you're about to receive valuable information that will help you succeed.

Focus on action

Achieving SPI success is difficult if process engineers aren't involved in the action phase of the IDEAL model, in which practitioners experience considerable fear and uncertainty.[2] During implementation, process engineers with change artistry[8] should work closely with software practitioners to resolve difficulties with both the new process and the process of change. You should, therefore, distribute available resources to ensure that process engineers and software practitioners are actively involved throughout the initiative, until the process has been successfully implemented.

Organize dedicated initiatives

SPI initiatives that target several units and projects often lack the resources and motivation needed to address the extremely complex change issues involved. Process engineers also sometimes prefer the early IDEAL model phases, in which they analyze and design the process, rather than the later phases, in which their challenge is to handle resistance and other barriers to change.[2] SPI initiatives supporting a single project or unit are less time consuming, require no compromises, and make it easier to create an open, collaborative relation-

About the Authors

Anna Börjesson is a software process improvement manager at Ericsson AB in Gothenburg, Sweden, and an industrial PhD student at the IT University in Gothenburg. Her software engineering skills cover such areas as CMM, Rational Unified Process, Rational Tool Suite, SW metrics, SW quality assurance, change management, SW diffusion and implementation theories, and SW process adaptation work. She received her M.Sc. in informatics from Gothenburg University. She's a member of the IEEE and ACM. Contact her at Ericsson AB, Lindholmspiren 11, 417 56 Gothenburg, Sweden; anna.borjesson@ericsson.com.

Lars Mathiassen is a professor of computer information systems at Georgia State University. His research interests include information systems and software engineering, with a particular focus on business process innovation. He received his Dr. Techn. in software engineering from Aalborg University. He's a member of the IEEE, ACM, and AIS and coauthor of *Computers in Context* (Blackwell, 1993), *Object Oriented Analysis & Design* (Marko Publishing, 2000), and *Improving Software Organizations* (Addison-Wesley, 2002). Contact him at the Center for Process Innovation, J. Mack Robinson College of Business, Georgia State Univ., PO Box 5029, Atlanta, GA 30302-5029; lmathiassen@gsu.edu; www.mathiassen.eci.gsu.edu.

ship between process engineers and software practitioners. You should, therefore, opt for dedicated SPI initiatives over generic ones whenever possible.

Our experiences at Ericsson focus on improvement tactics that affect SPI implementation success. You should, however, be cautious when you transfer the lessons to other software organizations. Factors such as process complexity, volume of initiative, organizational culture, and individual skills also affect SPI outcome. And, of course, there are always exceptional situations and those in which other approaches to process implementation are feasible. Ultimately, these are guidelines, not absolute truths. Our advice is to stay pragmatic but actively use these lessons to guide your SPI initiatives toward success.

References

1. W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.
2. B. McFeeley, *IDEAL: A User's Guide for Software Process Improvement*, tech. report CMU/SEI-96-HB-001, Software Eng. Inst., Carnegie Mellon Univ., 1996; www.sei.cmu.edu/pub/documents/96.reports/pdf/hb001.96.pdf.
3. R.W. Zmud, "An Examination of 'Push-Pull' Theory Applied to Process Innovation in Knowledge Work," *Management Science*, vol. 30, no. 6, 1984, pp. 727–738.
4. R.G. Fichman and C.F. Kermerer, "The Assimilation of Software Process Innovations: An Organizational Learning Perspective," *Management Science*, vol. 43, no. 10, 1997, pp. 1345–1363.
5. C. Argyris and D. Schön, *Organizational Learning*, Addison-Wesley, 1978.
6. J.S. Brown and P. Duguid, "Organization Learning and Communities-of-Practice: Toward a Unified View of Working, Learning and Innovation," *Organization Science*, vol. 2, no. 1, 1991, pp. 40–57.
7. R.B. Grady, *Successful Software Process Improvement*, Prentice Hall, 1997.
8. G.M. Weinberg, *Quality Software Management, Volume IV: Anticipating Change*, Dorset House, 1997.
9. L. Mathiassen, P.A. Nielsen, and J. Pries-Heje, "Learning SPI in Practice," *Improving Software Organizations: From Principles to Practice*, L. Mathiassen, J. Pries-Heje, and O. Ngwenyama, eds., Addison-Wesley, 2002, pp. 3–21.
10. S. Tryde, A.-D. Nielsen, and J. Pries-Heje, "A Framework for Organizational Implementation of Software Process Improvement in Practice," *Improving Software Organizations: From Principles to Practice*, L. Mathiassen, J. Pries-Heje, and O. Ngwenyama, eds., Addison-Wesley, 2002, pp. 257–271.
11. P. Fowler and M. Patrick, *Transition Packages for Expediting Technology Adoption: The Prototype Requirements Management Transition Package*, tech. report CMU/SEI-98-TR-004, Software Eng. Inst., Carnegie Mellon Univ., 1998.

Artikel 12 bij leereenheid 4

Measuring the ROI of Software Process Improvement
Rini van Solingen

IEEE Software , volume 21, issue 3, May 2004

Measuring the ROI of Software Process Improvement

Rini van Solingen, *LogicaCMG*

Software process improvement has been on the agenda of both academics and practitioners, with the Capability Maturity Model[1] as its de facto method. Many companies have invested large sums of money in improving their software processes, and several research papers document SPI's effectiveness. SPI aims to create more effective and efficient software development and maintenance by structuring and optimizing processes. SPI assumes that a well-managed organization with a defined engineering process is more likely to produce products that consistently meet the purchaser's requirements within schedule and budget than a poorly managed organization with no such engineering process. A sound process is, however, merely one prerequisite: it doesn't guarantee good products. With the amount of attention, literature, and investments focusing on SPI, the question regularly pops up whether these investments are worth their cost.[2,3] Surprisingly, we find only a limited number of industrial SPI publications that contain cost-benefit numbers and that measure ROI (see the "ROI Numbers for SPI" sidebar).

Software practitioners often say that they can't accurately calculate return on investment because they can't quantify software process improvement's benefits. On the contrary, we can measure benefits just as easily as we measure cost.

Analyzing SPI's ROI is relevant for

- Convincing managers to invest money and effort in improvement, and convincing them that SPI can help solve structural problems.
- Estimating how much effort to invest to solve a certain problem or estimating whether a certain intended benefit is worth its cost.
- Deciding which process improvement to implement first. Many organizations must prioritize due to timing and resource constraints.
- Continuing improvement programs. SPI budgets are assigned and discussed yearly, so benefits must be explicit and organizations must show sufficient ROI, or continuation is at risk.

ROI Numbers for SPI

You can calculate software process improvement's return on investment by dividing a financial representation of the benefits by a financial representation of the cost. So, an ROI of 5 implies that every invested dollar brings 5 dollars' profit. A limited number of publications contain concrete data for calculating the ROI of SPI. Table A presents an overview of the ROI numbers taken from experience reports in the literature. Not all reports use the formula (benefits – cost)/cost for calculating ROI. Some use benefits/cost or leave out the calculation used. When ROI values are high, the difference is relatively small to the benefit; it's more critical, however, if the ROI approaches 1.

Table A shows that organizations report an ROI of SPI ranging from 1.5 to 19 for every invested dollar (Capers Jones states that he generally observes an ROI between 3 and 30 to every invested dollar[17]). The average ROI is 7 and the median of the data is 6.6. Although any SPI undertaking's ROI depends on many influencing factors, it appears that a proper estimation for an SPI ROI lies between 4 and 10.

However, the literature contains limited evidence that these ROIs will occur when you use a specific SPI. The best we can attain with studies focusing only on process factors is strong evidence that SPI is associated with some benefits or that organizations could benefit from SPI activities.[18] Therefore, SPI's benefits will strongly depend on why an organization starts SPI in the first place—what are the intended benefits? Literature findings are diverse and distributed among software engineering's numerous business goals. Furthermore, different SPI approaches have different effects.[19,20]

Although the publications on SPI's costs and benefits are written by respected researchers and have been through severe reviewing processes, we must consider some limitations to the reported data. Specifically, we must consider the validity of these findings—how good are they and are they generically true?[18,21] For example, people often report only success stories, not failures, and it's impossible to know how many SPI attempts have failed.

References

1. J. Brodman and D. Johnson, "Return on Investment from Software Process Improvement as Measured by U.S. Industry," *CrossTalk*, vol. 9, no. 4, 1996, pp. 23–29.
2. M. Diaz and J. King, "How CMM Impacts Quality, Productivity, Rework, and the Bottom Line," *CrossTalk*, Mar. 2002, pp. 9–14.
3. S.A. Slaughter, D.E. Harter, and M.S. Krishnan, "Evaluating the Cost of Software Quality," *Comm. ACM*, vol. 41, no. 8, 1998, pp. 67–73.
4. J. Herbsleb et al., *Benefits of CMM-Based Software Process Improvement: Executive Summary of Initial Results*, tech. report CMU-SEI-94-SR-13, Software Eng. Inst., 1994.
5. D.K. Dunaway et al., *Why Do Organizations Have Assessments? Do They Pay Off?*, tech. report CMU/SEI-99-TR-012, Software Eng. Inst., 1999; www.sei.cmu.edu/publications/documents/99.reports/99tr012/99tr012abstract.html.
6. W. Humphrey, T. Snyder, and R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, vol. 8, no 4, 1991, pp. 11–23.
7. A. Goyal et al., "ROI for SPI: Lessons from Initiatives at IBM Global Services India," best paper at the 3rd SEPG Conf.: Software Excellence in the e-Economy, 2001; www.qaiindia.com/Conferences/SEPG2001/presentation.htm.
8. M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software*, vol. 14, no. 5, 1997, pp. 75–81.
9. K. Butler, "The Economic Benefits of Software Process Improvement," *CrossTalk*, vol. 8, no. 7, 1995, pp. 14–17.
10. J. Rooijmans, H. Aerts, and M. van Genuchten, "Software Quality in Consumer Electronics Products," *IEEE Software*, vol. 13, no. 1, 1996, pp. 55–64.
11. R. Dion, "Elements of a Process Improvement Program," *IEEE Software*, vol. 9, no. 4, 1992, pp. 83–85.
12. R. Dion, "Process Improvement and the Corporate Balance Sheet," *IEEE Software*, vol. 10, no. 4, 1993, pp. 28–35.
13. G. Yamamura and G.B. Wigle, "SEI CMM Level 5: For the Right Reasons," *CrossTalk*, vol. 10, no. 8, 1997; www.stsc.hill.af.mil/crosstalk/1997/08/index.html.
14. R.B. Grady and T. van Slack, "Key Lessons in Achieving Widespread Inspection Usage," *IEEE Software*, vol. 11, no. 4, 1994, pp. 46–57.
15. D. Reifer, A. Chatmon, and D. Walters, "The Definitive Paper: Quantifying the Benefits of Software Process Improvement," *Software Tech News*, Nov. 2002; www.dacs.dtic.mil/awareness/newsletters/stn5-4/toc.html.
16. L.G. Oldham et al., "Benefits Realized from Climbing the CMM Ladder," *CrossTalk*, vol. 12, no. 5, 1999, pp. 7–10.
17. C. Jones, "The Economics of Software Process Improvement," *Computer*, vol. 29, no. 1, 1996, pp. 95–97.
18. K. El Emam and L. Briand, *Cost and Benefits of Software Process Improvement*, tech. report ISERN-97-12, Int'l Software Eng. Research Network, 1997; www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-97-12.ps.gz.
19. D. Reifer, "Let the Numbers Do the Talking," *CrossTalk*, Mar. 2002, pp. 4–8.
20. T. McGibbon, *A Business Case for Software Process Improvement Revised: Measuring Return on Investment from Software Engineering and Management*, DACS tech. report SP0700-98-4000, 1999; www.dacs.dtic.mil/techs/roispi2.
21. S. Sheard and C.L. Miller, "The Shangri-La of ROI," *Software Productivity Consortium*, 2000; www.software.org/pub/externalpapers/Shangrila_of_ROI.doc.

Table A

ROI numbers in the literature

Context	Return on investment
Unknown*[1]	1.5 (Judith Brodman and Donna Johnson present ROI numbers from several cases: 1.5, 2.0, 4, *6*, 7.7, 10, 1.26, *5*. Those italicized are probably Tinker, Raytheon, and Hughes Aircraft.)
General Dyn. Dec. Systems[2]	2.2
BDN International[3]	3 (Sandra Slaughter and colleagues present four ROI numbers: 3.83, 3.65, 2.96, and 2.74.)
Unknown (U*)[4]	4
US Navy[5]	4.1
Unknown (W*)[4]	4.2
Hughes Aircraft[6]	5
IBM Global Services India[7]	5.5
Tinker Air Force Base[1]	6
Unknown (X*)[4]	6.4
Motorola[8]	6.77
OC-ALC (Tinker)[9]	7.5
Philips[10]	7.5
Raytheon[11,12]	7.72 (This number is calculated based on 6 projects. If the same calculation is followed for the reported 15 projects, this seems to result in an ROI of 4.)
Boeing[13]	7.75
Unknown (Y*)[4]	8.8
Unknown[1]	10
Hewlett-Packard[14]	10.4
Northrop Grumman ES[15]	12.5 (Donald Reifer and colleagues report an ROI by productivity gains of 1251% on a 5-year planning horizon.)
Ogden ALC[16]	19
Average	**7**
Median	**6.6**

*Character used to refer to the respective organization in Herbsleb et al.[4]

Calculating cost and benefits is a prerequisite for investment decision making. This is just as true for SPI as for any other investment.

- Surviving, because any investment in an organization should be valued against its return. Otherwise, money will likely be wasted and you risk bankruptcy in the long run.

Like any change in an organization, SPI is an investment for which the benefits should exceed the cost. One frequent argument in software practice is that measuring SPI's benefits is impossible, or at least difficult. I propose some pragmatic solutions on how to calculate cost and benefits and how to calculate the ROI.

Quantifying cost and benefits: Be pragmatic

Calculating cost and benefits is a prerequisite for investment decision making. This is just as true for SPI as for any other investment. Measuring cost and benefits doesn't have to be difficult. Being pragmatic and involving stakeholders makes quantification easy.

Measuring benefits is as easy as measuring cost

Organizations find it relatively easy to measure cost by measuring effort but have trouble measuring benefits. However, this is owing to a serious misunderstanding of cost measurement: costs are much broader than effort alone. For example, cost also involves other resources, such as office space, travel, and computer infrastructure. Usually when organizations calculate cost they use a fixed hour-rate that they assume acceptably approaches the cost's real value. This is a commonly accepted method. However, such a cost calculation is, in fact, an estimate. In itself, this method isn't wrong. It's a pragmatic agreement on how to approach actual cost with an acceptable accuracy level. However, if we accept that cost measurement is just a matter of estimating and agreeing on the procedure, why don't we do the same for benefits? If (just as with costs) we agree that approaching the actual value is sufficient and we agree on the estimation procedure, we can measure benefits to the same extent as we measure cost.

Measuring benefits is therefore just as easy as measuring cost. We only need to agree on the required accuracy level. Because ROI calculations for SPI don't usually need to be very accurate, we can easily measure benefits based on stakeholder involvement and estimation. Thus, we can incorporate explicit ROI calculations into SPI investments and evaluate whether the SPI activities were worth the effort.

ROI isn't a strong metric when calculating investments that exceed a one-year time span; in such cases, *net present value* is stronger. However, for this article, I consider only ROI, especially because any industrial investment should show short-term results within the same year.

ROI numbers ease decision making

As I mentioned, detailed ROI calculations aren't necessary. It's usually sufficient to know the ROI's relative value: is it positive, break-even, or negative? In most industrial organizations it isn't as important to know whether the ROI is 7.5 or 9.2; knowing whether it's positive and knowing its range (for example, between 5 and 10) is more than enough for most decision making. The sole reason for calculating ROI is to decide within a specific industrial context (and industry) where to invest the money. SPI is just one possible investment.

The ROI of SPI differs over different situations. For example, a company with severe quality problems at customer sites can obtain a much higher ROI from SPI than a company that wants to increase productivity, because the business benefits are higher in the first case. So, building a business case for SPI is always a specific task for a specific environment. Generic numbers on the ROI of SPI (see the sidebar) can help, but you should build the business case along the lines of the specific context, its goals, and its problems. We can't give a generic benchmark for SPI. However, when building the case for SPI in comparison to other investments, quantifying benefits and ROI will certainly help.

Furthermore, research has proven that humans make trade-off analyses continuously—if not on the basis of objective measurements then on intuition.[4] Making explicit ROI calculations is therefore crucial for SPI because it's an investment with significant cost and sometimes invisible benefits. The ROI should therefore be visible as well to avoid incorrect intuitive evaluations. Without numbers on SPI's cost, benefits, and ROI, it's impossible to properly decide whether SPI is worth its cost. Even if the overall SPI undertaking breaks even, local benefits might already be worthwhile. For example, if it saves a development team time, then it shortens time-to-market and developers can work faster with less pressure.

To show the ROI of SPI, we must focus on productivity and time-to-market impacts:

> *The true cost-benefits occur when projects finish earlier, allowing us to apply more engineering resources to the acquisition and development of new business.*[5]

Involve stakeholders for benefit estimation

When looking for a basis for measuring SPI benefits, consider that

> *Although intangible benefits may be more difficult to quantify, there is no reason to value them at zero. Zero is, after all, no less arbitrary than any other number.*[6]

So, using a certain number obtained from stakeholder estimation is better than just determining an intangible benefit as zero. Stakeholder involvement for benefit quantification seems logical. Stakeholders see benefits' impacts and values from specific viewpoints. Most people will agree that in practice, it's impossible to find a single person with a full overview on SPI benefits who can also express those in monetary terms.

Multiple stakeholders should therefore be involved. For example, if you know that SPI reduces time-to-market by two weeks, you can ask the marketing department what this will bring in financial values. You will get a number or an estimated range that you can use in your calculations. Also, don't forget to ask the project manager whether the project would have suffered from serious delays if the SPI actions had not been taken—a so-called "what-if-not" analysis.[6] If so, ask the marketing department what this delay would have cost—another benefit. It's important to include all these SPI benefits and to convert them to a financial value. After all, "money" is a measurement scale that most stakeholders understand.

An alternative to calculating pure cash-flow benefits is to ask those involved (for example, management) what a certain improvement is "worth." This means not just measuring the effort of the improvement activities but looking at that improvement's value and taking that value as the benefit.

> *Rather than attempting to put a dollar tag on benefits that by their nature are difficult to quantify, managers should reverse the process and estimate first how large these benefits must be in order to justify the proposed investment.*[6]

For example, if a manager states that his or her team is clearly more motivated owing to the SPI initiatives, ask the manager what price he or she would pay for that increased motivation. Ask the manager, for example, how many training days he or she would spend on staff to acquire this increased motivation. If it is, for example, five training days, you can quantitatively estimate this benefit: number of staff × number of days training × (daily rate of staff + daily fee of one-person training). As you see, the benefit is easy to quantify as long as there's agreement on how it's done.

Case studies

I've used the ideas just described in several projects and organizations. I present two case-studies as good practice on how you can do ROI analyses for SPI easily and pragmatically.

A goal, question, metric-based measurement program

My colleagues and I made an initial case for a GQM measurement program. This particular program took place in a systems development department (hardware and software) in an industrial company that produces and services systems for fuel stations. The software team developed an embedded-software product that controls a fuel pump and manages all fuel-transaction communications. This case involves a goal-oriented measurement program that addressed developer distortions (so-called *interrupts*).[7] The measurement program aimed to find out the reasons for developer interrupts and aimed to reduce them. During a three-month period, a six-person development team measured and improved their processes. Table 1 shows the measurements for this case.

When analyzing the cost, we find this improvement program's total cost was (320/1600) × US$100,000 = $20,000. We made the calculations using 1,600 productive engineering hours per year and a yearly cost of $100,000 per engineer. (The case took place in the Netherlands.) The software team's effort was 80 hours ($5,000) and the GQM measurement team's effort was 240 hours ($15,000). When considering the benefits, we measured that the software team saved 260 hours in engineering effort due to the improvements (reduced number of interrupts). The GQM measurement team saved 60 hours (from reusable material). These benefits related directly to the

Stakeholder involvement for benefit quantification seems logical. Stakeholders see benefits' impacts and values from specific viewpoints.

Table 1

Detailed measurements for Case 1's ROI calculation

Cost-benefits	Value (US$)	Explanation
Cost		
Engineering team's effort	$5,000	80 hours' effort expenditure in measurement-program-related tasks, measured from hour-registration system
GQM team's effort	$15,000	240 hours' effort expenditure for measurement program, measured from hour-registration system
Total cost	**$20,000**	
Benefits		
Effort saving due to less interrupts	$16,000	260 hours' effort saving during the measurement program due to a measured reduction of interrupts[7]
Effort saving reuse (GQM team)	$4,000	60 hours' effort saving due to reusable material on interrupt reduction
Total direct benefits	**$20,000**	
Early delivery due to effort saving	$100,000	One-week-early product delivery, measured from value the marketing manager indicated
Effort saving due to spin-off	$50,000	Effort saving during remainder of the year due to the reduction of interrupts
Increased quality awareness	$100,000	Increased focus on quality and time expenditure, both in the project as in other groups, measured from value for group manager (combination of buy-in and personal value)
Update of engineering documentation	$16,000	Some documentation was updated due to a measurable number of interrupts on these documents, measured from value for engineers
Total indirect benefits	**$266,000**	
Total benefits	**$286,000**	
ROI	**1:13**	

improvement program's objectives. Therefore, the software team's financial benefits were $16,000 and the GQM team's were $4,000. The software team's ROI is therefore 2, and the whole program broke even. We calculate the ROI by dividing the investment's profit by the investment: (benefit – cost)/cost.

However, when we consider the indirect benefits, made clear in the measurement program's feedback sessions and based on the project manager's conclusions, the benefits are higher:

- The project finished at least one week earlier than expected thanks to the measurements (according to marketing, a savings of at least $100,000).
- Documentation was updated on the basis of the measurement analysis, preventing at least 260 hours of interrupts (equivalent to $16,000).
- The software team's quality awareness and interruption awareness increased (which the project manager valued at at least $100,000).
- Interruptions in other projects decreased in the same department owing to increased awareness outside the department (valued at more than $50,000).

We can calculate the total benefits to be at least $286,000, making the software team's ROI 55—that is, ($286,000 – $4,000 – $5,000)/$5,000. The whole organization's ROI is therefore 13—($286,000 – $20,000)/$20,000.

Distinguishing between direct and indirect benefits supports the business case for SPI. The indirect benefits especially (those that are difficult to correlate directly to SPI efforts because they're generated from multiple initiatives) tend to have large financial benefits. Although quantifying those benefits requires some effort, it serves to explain to managers why SPI initiatives support business goals.

The CMM-based improvement program

The second case study presents the results of an ROI evaluation of an industrial SPI program. This program used the software CMM as a starting point for improvement and applied it pragmatically as a checklist for potential improvement actions. The organization develops and services a software simulation package that can execute virtual tests using finite-element modeling. Such simulations give production companies safety feedback on products that are still on the "drawing table." This package's market success is, in fact, mainly due to its high ROI. Imagine the savings for a manufacturer when discovering safety flaws during the design phase rather than the delivery phase.

This particular organization defines its improvement goals in terms of development throughput time, schedule accuracy, and customer satisfaction. After one year in the SPI program, the ROI was evaluated by the SPI

Table 2

Detailed measurements for Case 2's ROI calculation

Cost-benefits	Value (US$)	Explanation	Allocation (%)	Value
Cost				
Company effort	$35,000	305 person-hours with an average hourly fee of $115, measured from project accounting system	100	$35,000
External coaching	$15,000	External coaching hours from consulting company, measured from bills	100	$15,000
Total cost				**$50,000**
Benefits				
Process awareness	$20,000	Measured from value for R&D manager, through buy-in comparison: 5 days by external trainer	100	$20,000
Documented processes available	$160,000	V-model reflected in set-of procedures and standard work breakdown structure for projects; effort saving at least $4,000 per project, 40 projects per year, measured from value for R&D manager	50	$80,000
Documentation templates	$120,000	Buy-in value of good template: $1,000, 3 templates set-up, 40 projects per year, measured from value for R&D manager and engineers	25	$30,000
Best practices documented	$32,000	Effort saving of at least $800 per project, 40 projects, measured from value for engineers	25	$8,000
Requirements training followed	$16,000	Cost of requirements training in effort and external trainer, measured from project accounting system	25	$4,000
Project documentation updated	$5,000	Updated documentation based on findings, measured from value for R&D manager	100	$5,000
Total direct benefits				**$147,000**
Project management support	$650,000	Calculated from value for R&D manager and product manager of the overall set of project management actions (for example, traffic light progress monitoring, customer planning alignment, less late deliveries)	10	$65,000
Release on time	$180,000	Effort and cost saving from releasing on time: $30,000, 6 releases, calculated from value for R&D manager and product manager	25	$45,000
Role separation	$255,000	Effort saving of 1.5 person-years, due to role and responsibility separation, measured from value for R&D manager	75	$190,000
Total indirect benefits				**$300,000**
Total benefits				**$447,000**
ROI				**1:8**

consulting company. The approach was undertaken using the the pragmatic ideas proposed in the previous section. Available measurements were expanded with five stakeholder interviews (marketing and product manager, development manager, software engineer, test engineer, and release coordinator). These interviews indicated that the SPI program's main benefits were

- Process documentation (description of standard processes, definition of templates and best practices, and a group-wide process-web infrastructure)
- Progress monitoring (periodic reporting by progress metrics and "traffic light" indicators)
- Software engineering role and responsibility definitions
- Improved product documentation

Stakeholders quantified each of these benefits and were asked for effort savings, a value range (between minimum and maximum), or a purchase value ("What if you had to buy this change?"). Every case used the lowest value of the stakeholder numbers, implying that the calculated ROI number was a minimum. One specific addition was made by adding so-called *contribution percentages*. Many improvements couldn't be attributed solely to the SPI program because they resulted from multiple initiatives, so the contribution to the improvement was indicated with such a percentage. Take, for example, the benefit "best practices." Best practices would have probably been documented even without an SPI program. However, the R&D manager estimated that the SPI program had a partial contribution of about 25 percent, owing to the focus on best-practice capturing. In this example, only 25 percent of the value was measured as benefit. Table 2 shows the measurements for this case study.

About the Author

Rini van Solingen is a principal consultant at LogicaCMG and a professor at Drenthe University, the Netherlands, where he holds a chair in quality management and quality engineering. His research interests include making (software) quality manageable and transferring new technologies successfully into practice. He received his PhD in management science from Eindhoven University of Technology. He is a member of the IEEE Computer Society, the Dutch society for informatics (NGI) and SPIder (the Dutch software process improvement network). Please contact the author if you have access to experience reports that include ROI data on SPI that are not included in this article. Contact him at LogicaCMG, PO Box 8566, NL-3009-AN, Rotterdam, The Netherlands; rini.van.solingen@logicacmg.com.

In the first year, the SPI program cost $50,000. When measuring benefits, we distinguished between the SPI program's direct benefits (directly attributed to activities in the improvement program) and indirect benefits (results more indirectly attributed to the improvement program). The direct benefits were valued at $147,000. The indirect benefits were valued at $300,000. This was calculated from the separate values for project management and control ($65,000), the on-time release of the product ($45,000), and role and responsibility definitions ($190,000).

Based on these collected numbers, it was relatively easy to calculate ROI numbers, using the same formula as the previous case study. The direct ROI was 2 to every invested dollar and the total ROI (including both direct and indirect benefits) was 8. The respective interviewed stakeholders agreed on the numbers underlying these calculations. When presenting them to the complete software engineering team, however, the engineers indicated that they didn't recognize all presented values. Apparently, not everyone was aware of the overall improvements and impacts. We concluded that more intermediate communication on SPI activities and results should have occurred, instead of just one yearly ROI analysis. This could have improved common understanding of the improvement program's benefits for the department.

My colleagues and I will continue facilitating the quantification of cost and benefits for software engineering. We're convinced that more attention for building and evaluating business cases will bring the software engineering discipline to a higher professional level. Quantifying economic aspects should be part of this discipline. Our work focuses on developing approaches that work in practice while being based on sound scientific theories.

Pragmatic ROI calculations are feasible and easy. Furthermore, they open a discussion on SPI's cost, benefits, and ROI. Pragmatism is crucial—you must apply an approach for measuring cost and benefits that's simple and fast by involving stakeholders. You must also accept that estimating such cost and benefits might not be perfectly accurate, but accuracy isn't the main purpose. The purpose is to indicate value, to indicate whether cost and benefits are balanced, and to obtain an ROI number for communication purposes. Expressing cost, benefits, and ROI in financials is crucial. Different people in different roles always have trouble understanding each other's language. If one generic term exists for which people share perception, it's money.

Acknowledgments

I thank Egon Berghout from Groningen University and the *IEEE Software* reviewers for their valuable comments to an earlier version of this article. This article results from the Information Technology for European Advancement research project Moose (www.mooseproject.org).

References

1. M.C. Paulk et al., "Capability Maturity Model for Software," v. 1.1, tech. report SEI-CMU-93-TR-24, Software Eng. Inst., 1993; www.sei.cmu.edu/publications/documents/93.reports/93.tr.024.html.
2. K. El Emam and L. Briand, *Cost and Benefits of Software Process Improvement*, tech. report ISERN-97-12, Int'l Software Eng. Research Network, 1997; www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-97-12.ps.gz.
3. D.F. Rico, *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*, J. Ross Publishing, 2004.
4. L.R. Beach, *Image Theory: Decision Making in Personal and Organizational Contexts*, John Wiley & Sons, 1990.
5. M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software*, vol. 14, no. 5, 1997, pp. 75–81.
6. R.S. Kaplan, "Must CIM be Justified by Faith Alone?" *Harvard Business Rev.*, vol. 64, no. 2, 1986, pp. 87–95.
7. R. van Solingen, E. Berghout, and F. van Latum, "Interrupts: Just a Minute Never Is," *IEEE Software*, vol. 15, no. 5, 1998, pp. 97–103.

Artikel 13 bij leereenheid 4

A tale of two cultures
Rob Thomsett

Thomsett Company 1996, update 2011

A tale of two cultures

2011 Update: This is another paper, which is just as relevant in 2011 as it was when I first wrote it in 1996. As we discuss in other papers and books, the lack of understanding that there are two different cultures in every organization causes more difficulties in project management and sponsorship than almost any other factor. Rob

Two half-cultures do not make a culture
Arthur Koestler (Ghost in the Machine)

The current focus in organisations on re-engineering, down-sizing, strategic value analysis, total quality management and self-managing teams is an attempt to meet the challenges that have evolved from the "excesses of the 80's" and the global recession of the early 1990's. However, as discussed by Charles Handy (1989), Tom Peters (1992), John Huey (1993), John Byrne (1993) and many others, rather than isolated remedial actions, these concepts signal the emergence of a new paradigm of organisation structure and behavior.

Terms such as the virtual organisation, the network organisation and the unglued organisation are used to propose radical solutions for the senior executives of organisations facing a chaotic environment of social, legislative and technological change.

In computing, a similar paradigm change is beginning to emerge. Concepts such as object-oriented development, client-server architectures, information resource management, rapid application development, networks, ICASE and group-ware are seen by experts such as James Martin (1991), Ed Yourdon (1992), Peter Keen (1991) and others as the solution to the productivity, service and quality pressures being brought to bear on computing groups by their clients. Further, as discussed by Thomsett (1992a), the move from technically-oriented service to business-oriented service has forced many computing groups to re-evaluate their well established processes of strategic and project management.

However, the benefits of improved organisation structures and information technology infrastructures will be diminished unless organisations and their information technology service groups address and resolve a much more fundamental issue : the existence of two conflicting cultures within their organisation.

Project and process cultures: a hidden conflict

The process culture in organisations has existed since at least the Industrial Revolution. Formalising the factory culture that had emerged in the 1800's, Frederick Taylor (1947) and Max Weber (1947) laid the theoretical framework in the early 1900's for the bureaucratic model of organisations which is still the prevailing paradigm for organisations.

However, from the formalisation of project management as a discrete technical discipline in engineering during the Polaris Programme in the 1950's, the growth of an alternative project culture has been accelerated in most organisations through the impact of information technology. By the 1960's, people such as Paul Gaddis (1959), C.J. Middleton (1967) and Gerry Weinberg (1971) had begun to draw attention to the special nature of projects, project people and, in the case of Weinberg's seminal book, of project teams.

However, as people who had begun to experiment with teams and project cultures in the 1970's discovered, the predominant process culture negated many of the advantages of the emerging project culture (Rob Thomsett (1981)). For example, in a project culture the whole team needs to be rewarded for a good job. In a process culture, the boss is rewarded for the team's good work.

The clash between process and project culture has been ignored for many years with the project culture forced to co-exist within the process culture like some alien immigrant. In the emerging organisation paradigm, the project culture will become the dominant culture. The impact of this shift will have broader and far reaching impact than the latest management fad or technological fix.

The differences between the two cultures can be understood by examining three aspects: organisation structure, the control/authority structure and job design. There are clearly many other aspects of organisation culture such as behavioral norms, socialisation processes and professional attitudes, however, these tend to be dominated by the three aspects above.

A blast from the past: the process culture

The process culture is based on routinised work undertaken in a stable environment by people and technology that are inter-changeable and "plug-compatible". The primary and covert concern is that when a machine or person breaks down they must be quickly replaced so that the expected productivity requirements are not impacted and business as usual resumes. The processes or tasks undertaken in the culture are engineered through scientific analysis and time and motion studies; and, a system is designed to optimise the efficiency of the work. This factory mentality applies to all forms of process cultures whether they are car manufacturers, banks, insurance companies or hospitals.


Organisation structure

The organisation structure of a process culture is hierarchical and rigid. To quote Weber "The organisation of offices follows the principle of hierarchy ; that is, each lower office is under the control and supervision of a higher

one". Further, the organisation is divided into sub-hierarchies that are designed with specialised technical competencies. For example, in a bank there might be a Retail Division, a Corporate Division and an Overseas Division. In a government Education Department, there might be a Tertiary Division, a Secondary Division and an Industrial Training Division.

The key to this structure is the concept of legitimised ownership in all terms of the knowledge, data, practice and standards of an organisational unit. As many people have observed, this ownership often results in dysfunctional behavior such as protecting one's territory at all costs. As observed by Stafford Beer (1975), the more under threat the bureaucracy, the more its efforts become diverted from servicing its clients to preserving its own survival.

This protection goes beyond people and procedures. Many a naive information resource manager has experienced the strong sense of ownership of data by senior managers when attempting to introduce data-naming standards.

Weber's concept was of a technically competent hierarchy where the position was separated from the person occupying the position. "The office is filled by a free contractual relationship. Thus, in principle, there is free selection based on a sphere of competence" Weber (op cit). However, as many of us has experienced, this concept has been distorted over time by the personalisation of a position by an occupant who seeks to build and protect his or her turf to justify higher remuneration, organisational status and power. The psychological needs for social acceptance, esteem and reward exacerbate the inherent concepts of higher and lower in hierarchies. Richard Pascale (1990) provides a detailed description of the "turf" wars between the various functional divisions in Ford Motor Company in the early 1980's. He quotes a senior executive " It was civil war at the top. The question was never 'Are we winning against the Japanese?' but rather, are we winning against each other?".

For the past 30 years, computing has had to exist within the process culture and hierarchical organisations and, as a result, many major systems reflect the hierarchical divisions of the larger organisations. The inability of the Retail Banking system to share data with the Personal Loan system has been well understood by information resource managers. Similarly, many project managers have been caught in the turf wars between their various clients as they attempt to balance conflicting system requirements.

In part, the move to new paradigms of more organic organisation structures such as virtual networks and Handy's three arm (leaf) model is a long overdue reaction to the failing of hierarchical structures.

 Control/authority

It is in the areas of control and authority that the process culture is the most evident.

Structure and control are closely linked and the higher the position in the structure the more control and authority the occupant of the position has legitimately vested in him or her.

Simple examples of the linking of control/authority and structure are the rules in many organisations regarding finance and human resources. In many bureaucracies, in an attempt to increase control of the organisation, the authority for spending of funds has been raised higher in the organisation. It is typical for rules such as expenditures over a certain level, eg $50,000 have to be approved by the C.E.O., to be implemented to tighten control in the organisation. Similarly, executive level approval for new recruitment and use of contract people are very common in hierarchical organisations fighting to retain control over expenses and to maintain the status quo.

More subtle forms of bureaucratic control and authority include rules for travel, accommodation, equipment and overtime. As noted by Pascale (op cit), the inevitable outcome of these control processes is that the organisation becomes controlled by the finance and accounting experts. A senior Ford executive explained "Finance occupied the critical spot at the top of the pyramid".

In a process culture, technical elites also engage in the turf battles using their expertise as a basis for control and authority. Engineers and computing people use their expertise to determine strategic technical decisions; the technical feasibility of new products and projects; and, the choice of technical infrastructures. For example, many organisations have to work within rigid technical architectural dogma such as all IBM and no Unix or Apple Macs dictated by their internal computing groups.

As noted by Thomsett (1992a) and others, while the technical control is explained by the need for a service ethic and a need for consistency and standards, in many cases, the rigid control of technical infrastructure is nothing more than an exercise in bureaucratic control and authority by technocrats.

Many attempts over the past 15 years to restructure hierarchical organisations have failed to realise that the shifting of people and positions in an organisation chart does not alter the basic control and authority structures or the existence of "informal" organisation structures or networks that eventually undermine the expected outcomes of the restructure. The phrase "rearranging the deck chairs on the Titanic" has become associated with the endless restructuring of organisations without addressing the control and authority that people have accumulated over

many years.

 Job design

If organisation structure and the associated control/authority are the architectural framework for the process culture, the design of jobs is the day-to-day manifestation of the culture. The key to process job design, as noted by Taylor in 1910, is that "the initiative of workmen is obtained practically with absolute regularity".

Jobs in a process culture can be identified with the following attributes:

.. repeating tasks

The typical process job will repeat many times a day. In a bank, the teller would service a customer in two minutes or less. In some factories, scientific analysis as proposed by Taylor some 80 years ago reduced the individual task to one minute or less.

.. routinisation and standardisation

It is a key to process work that the same job is performed according to the routine and standards independent of who is performing it. As a result, a bank teller may move from branch to branch without retraining. Standard operating procedures are developed and used as a guide for training, evaluation and enforcement of standardisation.

.. variation reduced

In process work, any variation in performance is seen as a "defect". This covert value has become institutionalised in the Total Quality Movement. W. Edwards Deming (Walton (1986)) and Joseph Juran (1989) who are credited with founding the TQM movement both see special causes (ie a person acting in a non-standard manner) as a major cause of poor quality in process work.

.. clear performance measures

Process work is easily measured and there are clearly established performance standards that have to be met each period. Most process organisations have highly developed performance measurement systems that monitor the rate of completion of tasks. For example, the bank manager knows the number of clients that should be handled by a teller on an average day. Performance is based on completion of a task. In addition, most performance measures can be gathered within very short times frames. For example, some service jobs have real-time performance measuring systems.

.. precise scope

As a result of the years of scientific analysis and bureaucratic territory protection, process culture jobs have clearly defined scope or boundaries. A teller handles general enquiries and a housing loan expert handles loans. This demarcation is even more evident in the union culture where each trade has precisely defined boundaries and skills.

.. operates within status quo

The most significant attribute of process work is that it constitutes business as usual for the organisation. In other words, at the end of a normal day, the bank branch, its people and its tasks are unchanged. The next day will be the same as the day before and any change is institutionalised as a one-off project and undertaken by experts.

Process work and tasks are the life blood of a process culture. The very nature of the work provides consistency and predictability that is also manifest in the organisation's structure and control. People know where they fit, where they belong and what they have to do.

Although Taylor and Weber were seeking to simply obtain consistency at the highest level of ability, the process culture let consistency and standards become ends in themselves.

To quote Taylor "the principles of scientific management ... must in all cases produce far larger and better results ... than can be obtained under the management of initiative and incentive in which those on the management's side deliberately give a large incentive to their workman, and in return, the workmen respond by working at the very best of their ability ..."

Levels of process work

There are at least three distinct levels of process work. They differ by the amount of discretion and problem-solving required for the job.

Level 1: Assembly

This is the "purest" type of process work. It involves binary decision-making and little or no problem solving. Examples include, factory assembly work, routine office tasks such as photocopying and typing. The learning requirements are simple - usually less than a day.

Level 2: Administrative/basic service

This level of process work requires slightly more complex problem solving and decision-making. Though the decision making and problem solving are still extremely limited and scoped. Examples include fast-food service delivery, office filing and "house-keeping" and document formatting. The learning requirements for this level are more sophisticated and can be between 5 to 20 days in duration.

Level 3 Complex service provision/support

This level involves some complex problem solving but the scope of the problems is still limited. The majority of decisions required have precedence and some of the problems may require projects to be undertaken. Examples include supervision of process work, nursing and similar hospital work, computer and other complex technology support or maintenance work and sophisticated document layout. Learning requirements can be many months in duration.

Now for something completely different: the project culture

The project culture is the exact opposite of the process culture. While the concept of projects probably pre-dated the process culture (the construction of buildings, the conduct of wars and the mounting of major explorations can all be considered as projects), the formalisation of the project culture did not commence until the 1950's. Although elements of the project culture such as project plans were developed by people such as Henry Gantt in World War 1, the emergence of a formalised body of knowledge and a specific organisational model began in the 1950's in the US Defence Department. As outlined by Peter Morris and George Hough (1987), the Atlas Missile Programme and the Polaris Programme under Admiral Raborn were early examples of what is now recognised as the project culture.

 Organisation structure

Organisation structures in the project culture are still evolving. However, whereas the process culture is based on hierarchies, the project culture is based on project teams. So, in many ways, the organisation structure of the project culture reflects the micro-organisation structure of project teams. However, the primary design is one of dynamic networks rather than rigid hierarchies.

At first, the project culture adopted the more dominant model of organisation structure and, as a result, were mini-hierarchies. As discussed by Thomsett (1992b), the traditional project team structure consisted of a project manager who had project leaders reporting to him or her and technical specialists (analysts, designers, etc) reporting to the project leaders. However, as early as the 1970's, computing experts such as Weinberg (op cit) and Larry Constantine (1992) were challenging the effectiveness of the hierarchical structure and its ability to deal with the dynamic and participative nature of decision making in teams.

Similar questions as to the effectiveness of hierarchical team structures in other industries were being asked by the advocates of the socio-technical system model such as Fred Emery and Eric Trist (1961). In the 1960's, the work of Trist, Emery and others in coal mines, car factories and steel works laid the theoretical foundation for the self-directed team model that is now central to the project culture. In 1961, Emery and Trist predicted that the inflexibility of the process culture in responding to environmental changes could lead to its failure. "The very survival of an enterprise may by threatened by its inability to face up to such demands, as for instance, switching the main effort from production of processed goods to marketing..."

The emerging project culture organisational model is one of flexible team structures based on leadership as a series of roles rather than a position.

Work by Weinberg in his concept of ego-less teams, Constantine's Structured Open teams and Thomsett's X-team was based on the concept of leadership moving
between team members based on their specific skills, personality and roles.

In summary, there is not one preferred organisation structure but rather the organisation structure in the project culture is dynamic and depends on the organisation's mission, team members and the specific project. Two early models of the project culture organisation structure can be seen in Handy's Shamrock structure and Charles Savage's (1990) network design. Both models share the concept of a small strategic and policy cell (that may be hierarchical) and a flexible network of project teams.


Control/authority

Whereas, control and authority in the process culture are based on position in the hierarchy and formal organisation delegation of authority, control and authority in the project culture is more complex.

The fact that a manager in a process culture can review performance of his or her subordinates and make other major decisions regarding their careers gives legitimised control and authority to the manager. In the project culture, control and authority must be won through performance rather than given through position.

As discussed by Thomsett (1992c) and others, the complex communications, negotiations and decision-making required in a project culture are beyond most individuals and there is a need for full participation by project team members and project stakeholders (providers of service to the project team). As discussed later, the predominance of team work in the project culture challenges many concepts such as leadership, rewards, recognition, performance measurement that are well established in the process culture.

In essence, control and authority in the project culture will be based more on competency, expertise and information.

As discussed by Ken Gailbraith (1983), the move from objective power such as control of resources, finance and rewards that were the basis of the process culture to the more subjective sources of power such as information, expertise and legitimacy has been reshaping the political map of most countries. A similar shift in sources of power will occur in the move from process to project cultures.

In contemporary team models, leadership within a team and organisation moves from person to person depending on the specific competencies of the person and the specific issue facing the team. For example, when

planning a project which requires many service providers to cooperate , the person in the team with the best negotiation and communication skills may be the dominant team member. Once the project is underway, a team member with superior technical skills may move into the leader role. The move to self-directed or autonomous teams in manufacturing as documented by Richard Wellins, William Byham and Jeanne Wilson (1991) and others indicates that the majority of organisations who are adopting the project and team culture use the rotating team leaders model.

However, as discussed by Constantine (op cit), there are a variety of team structures varying from highly-structured teams based on specific expertise and roles to open team structures with no specific structure. The key is that the structure of the team is project-specific and team member-specific.

 Job design

The design of jobs in the project culture provide the clearest indication of the difference between the process and project culture. In fact, the attributes of jobs in the project culture are the exact opposite of the jobs in the process culture.

Jobs in a project culture can be identified with the following attributes:

.. non-repeating tasks

The typical project job task would vary between projects and would reflect the unique nature of the project. Further, the tasks in projects can last for a considerable time. For example, the tasks undertaken by a systems analyst in a project may last many months.

.. unique and entrepreneurial

While project work may follow some agreed processes (a development life cycle) such as requirements analysis, design, development and implementation, the specific outcomes of the work is project-specific. As a result, the focus in project work moves from standardising the output to standardising (where possible) the techniques and technology. For example, many companies have developed standard systems analysis techniques such as data modelling or process modelling but, the outcome of the product analysis depends on the specific system or product involved in the project and upon the expertise of the person who is undertaking the analysis.

.. variation amplified

Project work requires flexibility and creativity. As a result, the success of a project often depends on people breaking the rules and questioning the

status quo. Variation between individuals and teams in the project culture are expected and encouraged. Studies by Capers Jones (1991), Lawrence Putnam and Ware Myers (1992) and many others have revealed individual and team performance variations of 10 :1 and greater in information systems work. This has profound implications for the process culture as discussed later.

.. vague performance measures

Project work is extremely difficult to measure and evaluate. While the use of formal planning techniques such as work breakdown structures can provide a basis of partitioning project work into shorter activities, the assessment of process in project work tends to be subjective and true assessment of progress is dependent on completion and quality review of task outputs. Given that many projects involve hundreds of tasks that are inter-related, whether a project is progressing successfully or not can often not be measured until its completion. Indeed, for most project work, the real measure of success is not the *delivery* of outputs at the end of the project but the effectiveness of the *implementation* of those outputs and the *impact* of those outputs on the status-quo. As a result, the real measure of a project's success may not be known for some years.

.. dynamic scope

The definition of the scope of project work is a clear example of the dynamic nature of project work. The scope of a project is flexible and reflects the changing nature of organisation systems. While project management seeks to control variations in project scope it cannot stop the variation. As explored by Thomsett (op cit), Jones and others, it is inevitable for project scope and objectives to alter during the project development cycle. These changes are driven by external requirement changes and by the iterative and progressively more detailed processes of analysis and design.

.. changes status quo

The most significant attribute of project work is that it is designed to change the status quo. The development of a new product or service, the implementation of a new human resource programme, the installation of re-engineered work practices are typical organisational changes projects that have an impact beyond the project manager and team.

In other words, the project culture is designed for continuous change and the basic organisation structure and job design pattern is dynamic and innovative. When a project has implemented the initial product or system, a continuous process of enhancement and refinement is typical. The innovative nature of the project culture also relies on the creativity of

people who, in stark contrast to Taylor's vision, are working at "the very best of their ability".

Two cultures in conflict

For the past 30 years, these two cultures have coexisted with the project culture being mainly found in the remote and technically oriented areas of the organisation such as computing and engineering. However, with focus of senior management moving to radical organisation changes, the project culture has begun to emerge in all areas of the organisation. For example, two major Australian organisations have reported that over 200 projects are underway at any one time.

Given the vast differences between the cultures, it is easy to identify the conflict between them and the resulting confusion that many people are now experiencing.

Examples of this conflict can be found in the issue of project stakeholders, financial processes, reward systems and career paths.

 Stakeholder versus line reporting

A project stakeholder is a person who has to provide services to the project manager or a person who is expecting service from the project. For example, in the development of a new insurance product typical stakeholders to the project manager developing the new product would be agency, administration, marketing, advertising, actuarial, audit and computer specialists, together with the product clients.

In a project, the project manager must negotiate with the various stakeholders as most of the stakeholders will "belong" to other organisational areas within the traditional hierarchy. In turn, each stakeholder is responsible for seeking approval for their service provision to the project with their line manager.

However, should a stakeholder be unable to fulfil their obligations to the project manager, then the project manager is caught between the two cultures. To resolve the conflict, the project manager must first inform his or her line manager and the project sponsor (who may be from a different organisational area). Then the sponsor and/or line manager must then raise the conflict with the line manager of the stakeholder who must then raise the issue with the stakeholder under their responsibility. In all some five people at least are involved as the line management concerns must be balanced with the project concerns.

With ten stakeholders, the project manager must handle negotiations with up to twenty people to resolve disputes to ensure that the process culture's concerns with turf and boundaries are met. As noted by Thomsett (op cit),

this complexity and conflict is one of the major causes of project failure.

 Project finance versus process budgeting

Financial management in the process culture is based on clearly-defined cycles and accountable items. Calendar year, financial year, account receivables, balance sheets, inventories, fixed assets and zero-base budgeting are the life blood of traditional accounting. However, the need for process culture accountants to balance the books yearly is in clear conflict with the flexible budgeting required in the project culture.

Return-on-investment is the focus of the project culture finance model and, for many projects, the "balance sheet" cannot be done yearly, quarterly, or on any other fixed cycle. The balancing of the books begins with the implementation of the project's deliverable and ends when the return-on-investment time frame is ended. For example, in Company A the ROI cycle is two years and for Company B the ROI cycle is seven years. In Company B, the accounts for the project cannot be balanced until Year 7.

In other words, in the process culture the budget, broken into specific account items which are allocated to an organisational unit for a year. In a project culture, the budget is allocated to a project (that will cross organisational areas) and for the period of the project life-cycle (however long it is). Of course, the use of project management tracking techniques can provide "interim" results.

 Individual versus team rewards

The fundamental tenets of reward and recognition in the process culture are that the individual is rewarded based on performance and position. Managers get rewarded for the performance of their subordinates and, as reported by John Byrne (1993), it is typical that managers get rewarded disproportionally (as Byrne reports, the average gap between C.E.O. pay and their technical experts has grown from 19:1 in the 1960's to over 100 :1 in the 1990's).

As discussed by Thomsett, Weinberg, Constantine and many others, in a project environment, each team member's unique skills are part of a synergistic process where the whole is greater than the sum of the parts. If a project is successful all team members - project manager, technical specialists, support people and other stakeholders - are vital to the success and should share equally in, or at least participate in determining the allocation of, the rewards.

Further, the move to performance agreements in process cultures is also problematical in the project culture. Performance agreements are again based on individual performance. What is the effectiveness of a performance agreement for a project manager based on meeting project

objectives, budget and deadlines when the project manager is dependent on team members and stakeholders and their line managers outside his or her authority for meeting those performance criteria?

 Specialist/horizontal career paths versus vertical career paths

The conflict involving technical career paths in the process culture has been well documented in all specialist project areas such as engineering, computing and research. Promotion and career path in the process culture has an implicit span of control component. For example, as a person is promoted from Technical Expert Grade 3 to Senior Technical Expert Grade 1, he or she is expected to supervise more technical experts. The Peter Principle reported by Lawrence Peters and Raymond Hill (1969) is invoked and the person spends more time managing people than applying their technical expertise. Eventually, with rapid changes in techniques and technology, the STE Grade 1 finds their technical skills obsolete and, with all the management demands, he or she has no time to attend training programmes to maintain their technical skills.

In a project culture, the career of technical experts and project management experts becomes horizontal as documented by Handy (op cit). By moving from project to project, the people in a project culture continually enhance their skills and are rewarded by new skills, new challenges and rewards based on added value, knowledge and skills rather than position. In the project culture, a career focuses on the building of specific portfolios of knowledge, skills and experience rather than organisational empires.

Other conflicts between the two cultures can be seen in competency development (the orientation of human resource development towards supervision, management and bureaucratic processes rather than project-oriented competencies), physical building construction (where office layouts suitable for process work are inadequate for project and team work), human resource allocation (the supervisor determines the people for work rather than the project manager selecting the appropriate people) and, in organisation reporting systems (where executive information systems are process rather than project-oriented).

Can these cultures peacefully coexist?

In most organisations, the two cultures have not existed together well. As shown in Figure 1, the traditional process culture may show project groups such as Policy Development, Information Technology, Strategic Planning and others in the standard organisation chart. However, in terms of corporate politics and dynamics, the project groups are marginalised as "weirdos" or, as one senior operational manager, when referring to his organisation's policy development group, put it to us, "The university of the 7th Floor".

In the transition from process cultures to project cultures, the two cultures must learn to coexist. This coexistence will require some creativity, and, as a result, hybrid organisation structures and approaches have begun to emerge.

One common model is the streaming of technical experts into management or administrative and technical careers. This goes some way to avoid the "Peter Principle" as it allows those technical experts whose preference is for a career as a technical person to move to senior management positions labeled as Consultant or Technical with no requirement for the hierarchical responsibility of subordinates. In many projects, it is possible for the project manager to be earning less than the technical consultants who report in a project sense to the project manager. Clearly, this can lead to conflict as the process culture organisational status can become confused with the project culture authority vested in the project manager.

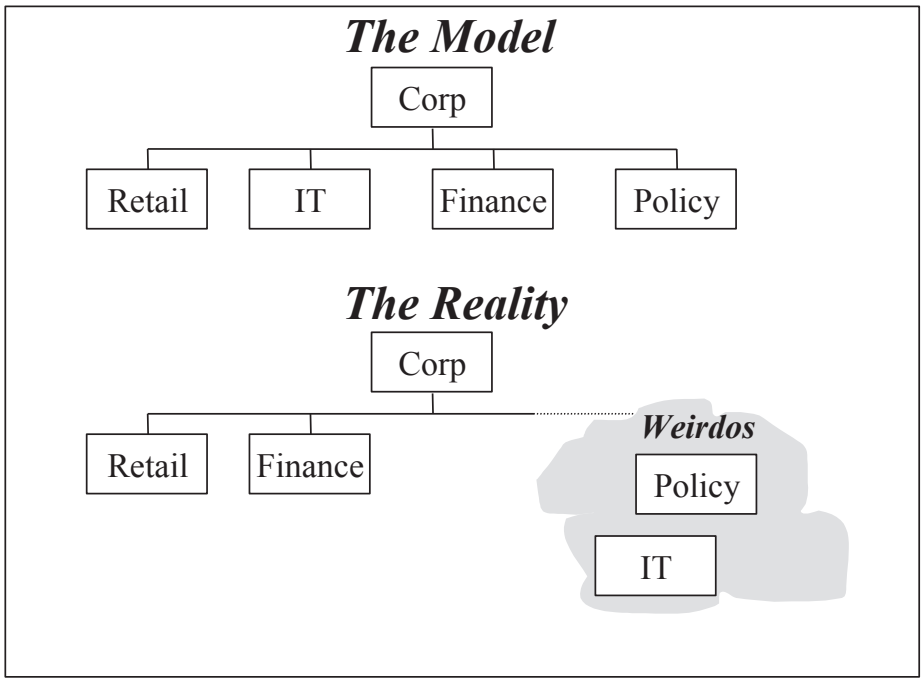


Fig. 1 - Process and project culture

The implementation of programme budgeting and activity-based costing (Brimson, 1991) are two attempts to break the time-based cycle of process culture accounting. With programme budgeting, a "trust" account is set up for the project which enables flexibility in the allocation of the funds and the reporting cycle. Activity-based costing attempts to avoid the use of generic project on-costing guidelines such as salary plus 155% on-costs (Thomsett (op cit)). For example, the equipment costs and other non-salary costs are identified project by project enabling tailored on-costing calculation. However, both these approaches are consistently compromised

by the long-established accounting systems and reporting structures built around the process culture accounting models.

The establishing of separate organisation units (often called skunkworks, independent business units, swat teams) is another common attempt to build hybrid structures. However, as discovered by IBM's PC group and many other isolated project cultures, the "re-entry" of their product into the process culture is exacerbated by the very fact that the product was developed by an organisationally remote group. This re-entry problem is also experienced by people seconded into project teams as project managers, business analysts and client representatives. The longer these people work in the project culture the harder it is for them to be accepted by their process culture colleagues and the harder it is for them to readjust to the process culture.

Other hybrid approaches such as the matrix structure (Struckenbruck (1989))and expert resource pools which attempt to provide the flexibility of skills required during the project development cycle with the hierarchical structures of the process culture, generally lack effectiveness as the people are caught between the two cultures. The matrix structure is consistently dealing with conflicts of interest between the technical specialists and their culture and the project manager and the process culture.

In computing, the concept of software engineering exhibits many of the features of the hybrid approach. Implicit in the concept of engineering is the Taylorist view of predictability and conformity. In other words, the ideal of software engineering is to minimise the variability of individuals and to maximise the use of repeatable, engineered processes. However, detailed studies by Jones (op cit), Putnam and Lawrence (op cit) and others reveal that computing work is highly creative and individualistic.

Even organisations that have structured and standardised their project development technology and techniques have found that data-modelling techniques cannot eliminate the 10 : 1 productivity variance between people. The inability of the process culture to reward and develop exceptional performance coupled with the software engineering model simply leads to a "levelling down" of expertise and performance as the high performers leave.

In summary, hybrid cultures are, at best, a temporary fix and, at worst, they result in confusion and conflict to both culture's disadvantage.

Where to now?

The emergence of the project culture and its associated dynamic structures, reward systems and leadership models is inevitable as the rate of change and competition increases for the remainder of the 1990's.

The shift from the process to project culture must be seen as an integrated and long-term process rather than a series of isolated initiatives such as business reengineering and self-directed teams. While these and other popular management concepts are part of a suite of enabling techniques, the culture shift will impact all existing components of the process culture including the service/business processes (the focus of reengineering), the accounting/finance procedures, the human resource/appraisal/reward processes, organisation and team structures, physical office design and so on.

Senior management must steer the culture shift and provide extensive education to all people in the areas of project management, team building, business analysis and other competencies well established in current project environments. In addition, they must integrate the implementation of strategic value analysis and planning, business process reengineering, activity-based accounting, competency-based human resource development, flexible reward systems, flexible information technology infrastructures, total quality management, project management and performance appraisal systems.

While this integrated approach is expensive, the cost of the culture shift is far outweighed by the expensive failures of the current process culture as giant organisations such as IBM, G.M. and other icons are outmaneuvered by smaller project culture organisations such as Microsoft and Intel.

At last, the project culture identified in the 1960's by pioneers such as Emery and Trist and Stafford Beer, and popularised later by Alvin Toffler (1980) and more recently by Peters and others, is slowly becoming the dominant paradigm. As highlighted by Thomsett (op cit), whereas Emery, Trist and Beer based their arguments on a value system oriented towards the enrichment of the working lives of people, the current focus is oriented towards a more fundamental value - survival!

However, while organisations and their clients will clearly reap substantial benefits, the real winners will be the people in the organisation who will join teams in dynamic and challenging projects and experience "working at the very best of their ability ..."

References

S. Beer, *Platform for Change*. New York, N.Y. John Wiley & Sons, 1975.

R.M. Belbin, *Management Teams: Why They Suceed or Fail*. London, Heinemann, 1981.

J.A. Brimson, *Activity Accounting*. New York, N.Y., John Wiley & Sons, 1990.

J.A. Byrne, "Executive Pay Ain't Over Yet", *International Business Week*, April 26, 1993.

J. A. Byrne, "The Virtual Corporation", *International Business Week*. February, 8, 1993.

L.L. Constantine, "Team-work Paradigms and the Structured Open Team", *Proceedings of the Embedded System Conference*. San Francisco, 1990.

F.E.Emery & E.A. Trist, "Socio-technical systems", *Systems Thinking*. London, Penguin Books, 1969.

P.O. Gaddis, "The Project Manager", *Harvard Business Review*. March/April, 1959.

J. K. Gailbraith, *The Anatomy of Power*. Boston, Ma., Houghton Mifflin, 1983.

C. Handy, The Age of Unreason. Boston, Mass., Harvard Business School Press, 1989

J. Huey, " Managing in the midst of chaos", *Fortune International*. Vol. 7, April 5, 1993.

C. Jones, *Applied Software Measurement*. New York, N.Y. McGraw Hill, 1991.

J.M. Juran, *Juran on Leadership for Quality*. New York, N.Y., Free Press, 1989.

P. G.W. Keen, *Shaping the Future : Business Design through Information Technology*. Boston, Mass., Harvard Business School Press, 1991.

J. Martin, *Rapid Application Development*. New York, N.Y., Macmillan, 1991.

C.J. Middleton, "How to Set Up a Project Organisation", *Harvard Business Review*. March/April, 1967.

P.W.G. Morris & G.H. Hough, *The Anatomy of Major Projects*. New York, N.Y., John Wiley & Sons, 1987.

R. Pascale, *Managing on the Edge*. Jondon, Penguin Books, 1990.

L.J. Peter & R. Hull, *The Peter Principle:Why Things Go Wrong*. New York, N.Y., William Morrow & Company, 1969.

T. Peters, Liberation Management. London, Macmillan, 1992.

L.H. Putnam and W.Myers, *Measures of Excellence*. Englewood Cliffs, N.J.,Prentice Hall, 1992.

C. M. Savage, *Fifth Generation Management*. Burlington. Ma., Digital Press, 1990.

L. Struckenbruck, *The Implementation of Project Management*. Reading, Ma., Addison-Wesley, 1989.

F. W. Taylor, "Scientific Management", *Organisation Theory* (ed D.S. Pugh).London, Penguin Books, 1971.

A. Toffler, *The Third Wave*. New York, N.Y., Bantam, 1980.

R. Thomsett, *People and Project Management*. Englewood Cliffs., N.J. Prentice Hall, 1981.

R. Thomsett, " The X-team : An innovative team structure for systems development", *American Programmer*. Vol 5. No 1., January 1992b.

R. Thomsett, "Computer and business professionals : An evolutionary perspective", *Professional Computing*, Australian Computer Society, July 1992a.

R. Thomsett, *Third Wave Project Management*. Englewood Cliffs., N.J. Prentice Hall,

1993c.

M. Walton, *The Deming Management Method*. New York, N.Y., Dodd, Mead & Company, 1986.

M. Weber, "Legitimate Authority and Bureauracy", *Organisation Theory* (ed D.S. Pugh).London, Penguin Books, 1971.

R.S. Wellins, W.C. Byham & J.M. Wilson, *Empowered Teams*. San Francisco, Jossey Bass, 1991.

G.M. Weinberg, *The Psychology of Computer Programming*. New York, N.Y. Von Nostrand Reinhold, 1971.

E.N. Yourdon, *Decline & Fall of the American Programmer*. Englewood Cliffs, N.J. Prentice Hall, 1992.

Artikel 14 bij leereenheid 5

Softwarekwaliteit, wat is dat?

Uit: *Softwarekwaliteit*
Fred J. Heemstra, Rob J. Kusters en Jos J.M. Trienekens (2002), pp. 79-97

6 Softwarekwaliteit, wat is dat?

6.1 Inleiding

Software wordt in steeds meer en in een steeds grotere diversiteit van toepassingsgebieden gebruikt. De juiste werking is vaak van wezenlijk belang voor het succes van een bedrijfsvoering en ook voor de veiligheid van mensen. Het goed specificeren en evalueren van softwarekwaliteit is noodzakelijk om kwaliteit te kunnen realiseren en te kunnen garanderen. In deel I zijn verschillende definities van kwaliteit van Garvin behandeld. In dit tweede deel van het boek wordt met name uitgegaan van Garvin's 'User Based' en 'Product Based' definities van kwaliteit, ook wel de gebruikersgerichte en de productgerichte benadering genoemd. We maken daarbij de volgende kanttekeningen:

- Het begrip 'User Based' komt ter sprake wanneer het begrip kwaliteitsbehoefte en het van daaruit afleiden van kwaliteitseisen voor software wordt behandeld.
- 'Product based' staat centraal wanneer softwarekwaliteit wordt opgedeeld in een aantal kwaliteitskenmerken.
- Verder kan worden gesteld dat ook Garvin's definitie 'Value Based' een rol speelt ofwel de waardegerichte benadering van kwaliteit, met name wanneer wordt gesproken over het stellen van prioriteiten voor te specificeren en te realiseren kwaliteitskenmerken.

Dit hoofdstuk heeft de volgende structuur. Paragraaf 6.2 gaat in op kwaliteitskenmerken van software. Kwaliteitskenmerken zijn van belang voor ontwikkelaars om door middel van een goed ontwerpproces kwaliteit aan software te kunnen meegeven. Anderzijds zijn kwaliteitskenmerken van belang voor softwaregebruikers om te kunnen aangeven waar voor hen softwarekwaliteit door wordt bepaald. Goede definities van kwaliteitskenmerken zijn nodig om kwaliteit bespreekbaar en hanteerbaar te maken. Paragraaf 6.3 beschrijft de 'state of the art' van het definiëren van softwarekwaliteit aan de hand van de ISO/IEC-standaard. In paragraaf 6.4 wordt beschreven dat afhankelijk van de situatie software op verschillende wijze kan worden beschouwd, en afhankelijk daarvan ook kwaliteitskenmerken kunnen variëren. Paragraaf 6.5 sluit het hoofdstuk af met conclusies.

6.2 Softwarekwaliteit, verschillende opvattingen

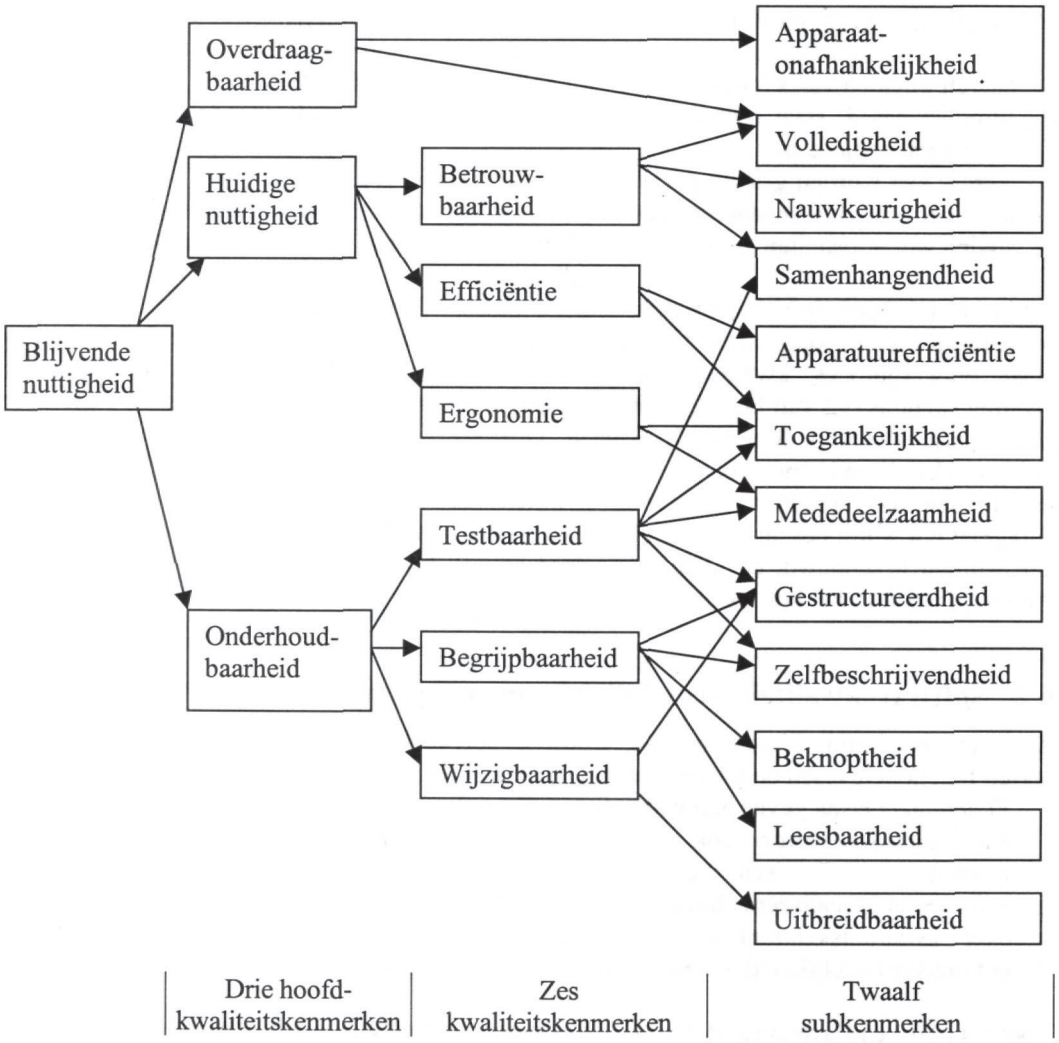
Zoals in hoofdstuk 3 werd gesteld, wordt in softwareontwikkeling onderscheid gemaakt tussen functionele eisen en kwaliteitseisen.

- *Functionele eisen* geven aan **wat** software moet doen.
- *Kwaliteitseisen* worden ook wel aangeduid met de term niet-functionele eisen. Bedoeld worden specifieke eisen, die betrekking hebben op de manier waarop functionaliteit van software is gerealiseerd (**hoe**). Wanneer voldaan is aan kwaliteitseisen beschikt software over kwaliteitskenmerken. Voorbeelden van kwaliteitseisen zijn betrouwbaarheid, gebruiksvriendelijkheid, responsesnelheid, en onderhoudbaarheid.

Boehm (1976) en Cavano en McCall (1978) kunnen in de software-industrie worden beschouwd als vertegenwoordigers van 'het eerste uur' als het gaat om het beschrijven van kwaliteitskenmerken. In hoofdstuk 3 werd reeds daaraan gerefereerd. Softwarekwaliteit wordt door deze auteurs beschreven aan de hand van een goed gedefinieerde verzameling kwaliteitskenmerken.

Verschillende andere onderzoekers hebben in de afgelopen decennia voortgebouwd op de eerste inzichten. Genoemd kunnen worden Willmer (1985), Deutsch en Willis (1987) en Bemelmans (1998). We zullen in het navolgende de kwaliteitsopvattingen van deze auteurs bespreken waarbij we in eerste instantie nog niet ingaan op de precieze betekenis van de afzonderlijke kwaliteitskenmerken en hun subkenmerken. Bij de bespreking van de verschillende opvattingen zullen we steeds weer de volgende drie invalshoeken hanteren:

- *wat staat centraal in de kwaliteitsopvatting*? We bespreken hier kort de verzameling kwaliteitskenmerken, de omvang en de structuur van die verzameling.
- *wie bepaalt kwaliteit*? Aan de orde komen de partijen die binnen de kwaliteitsopvatting worden verondersteld gebruik te maken van de kwaliteitskenmerken.
- *hoe wordt kwaliteit gerealiseerd*? Kort wordt aandacht besteed aan de manier waarop kan worden omgegaan met de kwaliteitskenmerken om kwaliteit in software 'in te bouwen'.



Figuur 6.1: Kwaliteit volgens Boehm et al (1976)

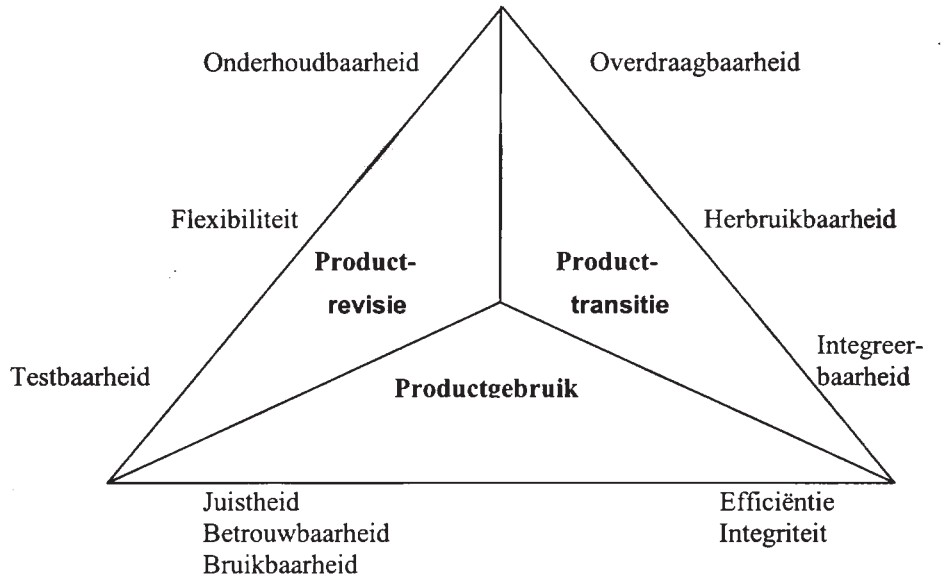
6.2.1 Kwaliteit volgens Boehm en Cavano en McCall (1976)

Wellicht het meest bekend op het gebied van kwaliteit in de software-industrie is de 'boom'-structuur van kwaliteitskenmerken van Boehm, zie figuur 6.1.

Wat staat centraal in de kwaliteitsopvattingen van Boehm, Calvano en McCall?

De genoemde structuur voor softwarekwaliteit staat bekend als de 'boom van Boehm' maar is in feite een netwerkstructuur (de vorm van figuur 6.1 geeft dat al aan). Het is de eerste poging in de literatuur om het begrip softwarekwaliteit hanteerbaar te maken. De 'boom' bestaat uit drie hoofdkwaliteitskenmerken, zes kwaliteitskenmerken en een groot aantal subkenmerken. Uitgangspunt is het begrip 'blijvende nuttigheid' dat is onderverdeeld in de drie hoofdkenmerken 'overdraagbaarheid', ofwel de verplaatsbaarheid van software, 'huidige nuttigheid', ofwel geschiktheid voor gebruik, en onderhoudbaarheid. Uit deze drie hoofdkenmerken van kwaliteit worden vervolgens langs hiërarchische weg kwaliteitskenmerken, zoals bijvoorbeeld betrouwbaarheid, gebruiksvriendelijkheid en begrijpbaarheid afgeleid en hieruit weer subkenmerken van kwaliteit, zoals bijvoorbeeld nauwkeurigheid en leesbaarheid.

Cavano en McCall kijken iets anders tegen kwaliteit aan. Zij maken onderscheid in kwaliteitskenmerken vanuit drie invalshoeken, te weten 'product operations' ofwel de werking van software, 'product revision' ofwel het aanpassen van software, en 'product transition' ofwel het (laten) evolueren van software naar een volgende generatie. Binnen elk van deze invalshoeken worden vervolgens kwaliteitskenmerken gedefinieerd, zie figuur 6.2.



Figuur 6.2: Kwaliteit volgens Cavano en McCall (1978)

Cavano en McCall maken voor het onderscheiden van kwaliteitskenmerken gebruik van de levenscyclus van software. Afhankelijk van de fase waarin software zich bevindt, of die software doorloopt, zijn verschillende kwaliteitskenmerken relevant. We merken op dat Cavano en McCall naast de invalshoeken voor softwarekwaliteit en de kwaliteitskenmerken ook een taal

hebben geformuleerd die zo goed als mogelijk begrijpelijk is voor gebruikers. Dit in tegenstelling tot de kwaliteitskenmerken in de 'boom van Boehm' die meer in ontwerperstaal zijn beschreven.

Wie bepaalt kwaliteit in de kwaliteitsopvattingen van Boehm, Calvano en McCall?

Op grond van de overwegend technische terminologie die wordt gehanteerd, kan worden gesteld dat de beide besproken benaderingen zich richten zich op het ontwerp ofwel de (technische) realisatie van kwaliteitskenmerken in software. De ontwerper wordt voorzien van een eenduidig begrippenkader in zijn streven naar kwalitatief goede software.

Hoe wordt kwaliteit bewerkstelligd in de kwaliteitsopvattingen van Boehm, Calvano en McCall?

Subkenmerken worden afgeleid uit kwaliteitskenmerken. Hoge kwaliteit lijkt afhankelijk te zijn van het aantal (sub)kenmerken dat software tijdens ontwikkeling meekrijgt. De subkenmerken, die van een sterk technische signatuur zijn, vormen het uitgangspunt voor te nemen maatregelen en een basis voor het inrichten van het ontwerp- en programmeerwerk.

6.2.2 De kwaliteitsopvatting van Willmer (1985)

Willmer (1985) gaat een stap verder met het specificeren van kwaliteitskenmerken dan voorgaande auteurs. Zij geeft een model waarin kwaliteitskenmerken, opgesteld door klanten, zijn gerelateerd aan kenmerken die ontwerpers in software kunnen aanbrengen, zie figuur 6.3.

Productkenmerken van de ontwikkelaars / makers → ↓ **Kwaliteitskenmerken van klanten/afnemers**	- Taalgebruik (duidelijke naamgeving, lengte zinnen, gebruik goto, enz.)	- Controle mogelijkheden in de software (access control)	- Mate van standaardisatie	- Mate van structurering	- Documentatie	- Visualisering
Begrijpbaarheid	X	X	X	X		X
Veranderbaarheid	X	X	X		X	X
Overdraagbaarheid op andere hardware		X	X		X	X
Onderhoudbaarheid	X	X	X			X
Te koppelen met andere software		X	X	X		
Mogelijkheid van hergebruik	X	X	X			
Testbaarheid	X	X	X			X
Efficiëntie van de software			X		X	X
Beveiliging (autorisatie gebruik)					X	
Correctheid (vrij van fouten)				X		
Gevoeligheid voor fouten					X	
Herstartbaarheid na foutoptreden					X	

Figuur 6.3: Kwaliteitskenmerken en productkenmerken (Willmer 1985)

Ook de kwaliteitsopvatting van Willmer wordt weer vanuit de drie invalshoeken *wat*, *wie* en *hoe* toegelicht.

Wat staat centraal in de kwaliteitsopvatting van Willmer?

De twee verzamelingen van kenmerken die door Willmer worden onderscheiden zijn enerzijds een verzameling softwarekwaliteitskenmerken vanuit het perspectief van softwaregebruik en anderzijds een verzameling softwarekenmerken vanuit ontwikkelperspectief. Doel van dit onderscheid is de communicatie tussen gebruikers van software en ontwikkelaars te verbeteren. Gebruikerseisen betreffende de kwaliteit kunnen aan de hand van een goed gedefinieerde terminologie eenduidig worden vertaald naar een verzameling 'productkenmerken.

Wie bepaalt kwaliteit in de kwaliteitsopvatting van Willmer?

Twee (hoofd)groepen betrokkenen worden onderscheiden, te weten de ontwikkelaars en de gebruikers of de klanten. Gebruikers dienen hun behoeften aan kwaliteit onder woorden te brengen. Ontwikkelaars nemen vervolgens op basis daarvan maatregelen om bepaalde softwarekenmerken te realiseren. Op de horizontale as in de figuur zijn een aantal (n) productkenmerken in de taal van ontwikkelaars weergegeven (bijvoorbeeld 'taalgebruik in de software', 'mate van structurering' etc.). Op de verticale as staan een aantal (m) kwaliteitskenmerken in de taal van de gebruikers of de klanten, bijvoorbeeld: 'begrijpbaarheid', 'herstartbaarheid na foutoptreden'. Willmer's model kan worden beschouwd als een aanzet om de twee werelden van ontwikkelaars en gebruikers naar elkaar te brengen.

Hoe wordt kwaliteit bewerkstelligd in de kwaliteitsopvatting van Willmer?

Willmer wijkt af van het opdoen (hiërarchisch) afleiden van subkenmerken uit geaggregeerde of hoofdkwaliteitskenmerken zoals dat door Boehm wordt beschreven. Door een netwerk van (n:m)-relaties te beschrijven tussen twee aparte verzamelingen wordt de vertaling van kenmerken uit de ene naar de andere verzameling ondersteund. Uiteraard spelen hierbij tal van beperkingen en randvoorwaarden een rol, die zowel van organisatorische als van technische signatuur kunnen zijn. Bijvoorbeeld de standaardisatie die een ontwerper wenst te hanteren moet vaak afgestemd worden op geldende organisatiestandaards, en mogelijkheden voor visualisering (grafische weergave-mogelijkheden, kleuren en geluid) hangen vaak samen met de mogelijkheden van de technologie waarover men beschikt.

6.2.3 De kwaliteitsopvatting volgens Bemelmans (1998)

Door Bemelmans (1998) wordt onderscheid gemaakt tussen gebruikseisen, beheereisen en eisen met betrekking tot informatie (zie figuur 6.4).

Wat staat centraal in de kwaliteitsopvatting van Bemelmans?

Het onderscheid tussen gebruikseisen en beheereisen komt in grote lijnen overeen met het onderscheid tussen deelverzamelingen van kwaliteitskenmerken in de eerder genoemde opvattingen van Boehm en Cavano & McCall. Belangrijk verschil vormt echter de aparte deelverzameling van kwaliteitseisen betreffende de informatie die door software wordt gegenereerd. Gesteld wordt dat de kwaliteitseisen die aan informatie kunnen worden gesteld

vaak een uitgangspunt vormen voor het formuleren van kwaliteitseisen over het gebruik en het beheer van software.

Wie bepaalt kwaliteit in de kwaliteitsopvatting van Bemelmans?

De kwaliteitseisen met betrekking tot informatie zijn sterk bedrijfscontext- of organisatiegebonden. Deze eisen komen over het algemeen op intersubjectieve wijze tot stand, dat wil zeggen door gebruikers in gezamenlijk overleg. Verder wordt in deze kwaliteitsopvatting, in tegenstelling tot de voorgaande opvattingen, benadrukt dat beheer ook eisen stelt aan softwarekwaliteit. In plaats van het ontwerpperspectief, dat in de twee voorgaande kwaliteitsopvattingen werd genoemd, wordt in Bemelmans' opvatting het veel bredere gebruik- en beheerperspectief gehanteerd.

Hoe wordt kwaliteit bewerkstelligd in de kwaliteitsopvatting van Bemelmans?

Voor wat betreft het realiseren van softwarekwaliteit komt deze opvatting in grote lijnen overeen met de twee voorgaande opvattingen, namelijk kwaliteitseisen dienen te worden vertaald naar bepaalde kenmerken. Bemelmans richt zich met name op het 'voortraject' ofwel het afleiden van kwaliteitseisen uit de karakteristieken van een organisatie. Gesteld wordt dat kwaliteitseisen in tegenstelling tot functionele eisen in mindere mate kunnen worden afgeleid van de te ondersteunen operationele processen, taken of procedures. Veel beter kan worden uitgegaan van de kenmerken van het type organisatie, en bijvoorbeeld de manier waarop processen worden aangestuurd (Bemelmans 1986). In lijn met deze opvatting worden later in dit deel van het boek benaderingen voor het specificeren van kwaliteitseisen beschreven (zie hoofdstuk 6).

Eisen voor het product INFORMATIE	**Eisen voor het SYSTEEM**	
	Eisen voor het GEBRUIK	**Eisen voor het BEHEER**

- Juistheid
- Volledigheid
- Actualiteit
- Controleerbaarheid
- Nauwkeurigheid
  - aggregatie
  - selectie
	- Tijdigheid
	- Integriteit
	  - juistheid
	  - volledigheid
	  - betrouwbaarheid
	- Security / beveiliging
	- Doelmatigheid
	- Effectiviteit
	- Gebruiksvriendelijkheid
		- Flexibiliteit
		- Onderhoudbaarheid
		- Testbaarheid
		- Portabiliteit
		- Inpasbaarheid in de technische infrastructuur
		- Herbruikbaarheid

Figuur 6.4: Kwaliteitseisen vanuit drie invalshoeken Bemelmans (1998)

6.2.4 Conclusies

Het bovenstaande laat zien dat verschillende auteurs in de afgelopen decennia hebben gestreefd naar een zo goed mogelijke beschrijving van softwarekwaliteit. De overeenkomsten en verschillen van de verschillende opvattingen hebben de roep om een eenduidig en operationeel begrippenstelsel voor softwarekwaliteit versterkt. Alleen standaardisatie lijkt tegemoet te kunnen komen aan die behoefte. In het navolgende bespreken we dan ook het werk dat de afgelopen jaren werd verricht op het terrein van standaardisatie van softwarekwaliteit binnen de internationale standaardisatiegemeenschap ISO/IEC.

6.3 Softwarekwaliteit: op weg naar een standaard (ISO/IEC 9126)

In de internationale standaardisatiegemeenschap zijn de afgelopen jaren verdere stappen gezet op het terrein van het specificeren, het realiseren en het evalueren van softwarekwaliteit. Binnen de ISO/IEC 9126 standaard wordt een onderscheid gemaakt tussen interne softwarekwaliteit, externe softwarekwaliteit en softwarekwaliteit-in-gebruik. De standaard wil voor het omgaan met softwarekwaliteit een consistente terminologie bieden en is bedoeld voor uiteenlopende groepen praktijkmensen, zoals betrokkenen bij selectie en inkoop van software, ontwikkeling van software, het specificeren van kwaliteitseisen, het gebruiken van de software, de evaluatie van software, en onderhoudsdeskundigen. Voorbeelden van activiteiten waarbij gebruik kan worden gemaakt van de ISO/IEC 9126 standaard zijn:

- het identificeren van kwaliteitseisen
- het valideren van een definitie van eisen
- het bepalen van software-ontwerpdoelstellingen
- het identificeren van testdoelstellingen
- het vaststellen van criteria voor kwaliteitsborging
- het bepalen van de acceptatiecriteria van eenmaal gerealiseerde software.

Binnen ISO/IEC9126 worden twee delen onderscheiden, respectievelijk

- een deel waarin de begrippen interne en externe kwaliteit worden behandeld en
- een deel waarin kwaliteit-in-gebruik nader wordt beschreven.

Met name met het recentelijk ontwikkelde begrip kwaliteit-in-gebruik wordt getracht nadrukkelijker aan te sluiten bij de dagelijkse praktijk van een klant of softwaregebruiker.

6.3.1 Interne en externe kwaliteit

Het eerste deel van de ISO-standaard behandelt de begrippen interne en externe kwaliteit en de relaties daartussen. Met interne en externe kwaliteit worden in feite het ontwikkelperspectief en het gebruiksperspectief op softwarekwaliteit bedoeld. Interne kwaliteit is de kwaliteit die door een ontwikkelaar gerealiseerd kan worden door bepaalde maatregelen te nemen tijdens het ontwikkelproces en door bepaalde kenmerken in software in te bouwen. Externe kwaliteit is de kwaliteit zoals die door gebruikers of klanten wordt gewenst, geëist of ervaren. Interne en externe kwaliteit zijn eigenlijk de twee zijden van de 'medaille van softwarekwaliteit'. Voor deze softwarekwaliteit worden zes zogeheten 'hoofd'-kwaliteitskenmerken beschreven, te weten functionaliteit, betrouwbaarheid, gebruiksvriendelijkheid, efficiëntie, onderhoudbaarheid en portabiliteit. De definities luiden als volgt.

Functionaliteit
De mate waarin software in functies voorziet die voldoen aan gestelde en impliciete behoeften, wanneer de software wordt gebruikt onder gespecificeerde omstandigheden.

Betrouwbaarheid
De mate waarin software een bepaald niveau van werking en prestatie biedt, wanneer de software wordt gebruikt onder gespecificeerde omstandigheden.

Gebruiksvriendelijkheid
De mate waarin software inzichtelijk is, men zich de software eigen kan maken, de software gebruikt kan worden, en aantrekkelijk is voor een gebruiker, wanneer de software wordt gebruikt onder gespecificeerde omstandigheden.

Efficiëntie
De mate waarin software de juiste prestatie (performance) biedt, in relatie tot de hoeveelheid middelen die worden gebruikt, onder gespecificeerde omstandigheden.

Onderhoudbaarheid
De mate waarin software kan worden gemodificeerd. Modificaties omvatten correcties, verbeteringen of aanpassingen van de software aan veranderingen in de omgeving, en in de eisen en de functionele specificaties.

Portabiliteit
De mate waarin software kan worden overgebracht van de ene (hardware)omgeving naar een andere.

Elk van deze hoofdkenmerken is zorgvuldig gedefinieerd en vervolgens verder opgedeeld in subkenmerken (appendix 2).

Voor elk van de subkenmerken worden door ISO/IEC 9126 vervolgens twee verzamelingen metrieken aangereikt. Een verzameling zogeheten interne metrieken is bestemd voor de ontwikkelaar om metingen te verrichten aan de software, met name tijdens ontwikkeling. Aan de hand daarvan kan worden bepaald of de inspanningen om kwaliteit te realiseren succesvol verlopen. De verzameling externe metrieken kan door gebruikers worden gehanteerd om vast te kunnen stellen of software werkt en presteert zoals is beloofd en/of afgesproken. Externe softwarekwaliteit heeft namelijk betrekking op het gedrag van de software zoals die in de praktijk wordt gebruikt. Voor voorbeelden van interne en externe metrieken voor softwarekwaliteit wordt verwezen naar hoofdstuk 9.

Uiteraard is het de bedoeling dat de externe softwarekwaliteit het resultaat is van de interne kwaliteit zoals die is gerealiseerd door een ontwikkelaar. De relaties zijn echter uiterst complex en de ISO/IEC standaard waagt zich dan ook niet aan een analyse en beschrijving daarvan.

6.3.2 Kwaliteit-in-gebruik

Het tweede deel van de standaard beschrijft het begrip kwaliteit-in-gebruik. Kwaliteit-in-gebruik is onderverdeeld in vier kenmerken, te weten effectiviteit, productiviteit, veiligheid en tevredenheid.

De definitie van kwaliteit-in-gebruik luidt:

> *De mate waarin software een specifieke groep van gebruikers in een specifieke gebruiksomgeving in staat stelt om hun doelstellingen op effectieve en productieve wijze te bereiken, en op een veilige manier en naar tevredenheid.*

Ook de vier kenmerken van kwaliteit-in-gebruik zijn zorgvuldig gedefinieerd en van metrieken voorzien. We geven de definities in het navolgende. Voor voorbeelden van metrieken voor kwaliteit-in-gebruik wordt verwezen naar hoofdstuk 9.

Effectiviteit
De mate waarin software gebruikers in staat stelt om hun doelstellingen te realiseren op een nauwkeurige en volledige manier

Productiviteit
De mate waarin software gebruikers in staat stelt om de juiste hoeveelheid aan middelen aan te wenden in relatie tot de effectiviteit die in een bepaalde gebruiksomgeving dient te worden gerealiseerd.

> N.B.: middelen hebben bijvoorbeeld betrekking op 'tijd om een taak te verrichten', de 'gebruikersinspanning', 'materialen' of de 'financiële kosten van gebruik'. In deel IV wordt op deze zaken nader ingegaan.

Veiligheid
De mate waarin door software wordt voorzien in acceptabele risiconiveaus voor wat betreft gevaar voor mensen, bedrijfsschade en schade aan eigendom in een specifieke gebruiksomgeving.

> N.B.: risico's hebben hier met name betrekking op de mate waarin wordt afgeweken van de eerder genoemde zes hoofdkwaliteitskenmerken.

Tevredenheid
De mate waarin software in een bepaalde gebruiksomgeving tevredenheid bij gebruikers bewerkstelligt.

> N.B.: tevredenheid is de reactie van gebruikers op de interactie met de software, en omvat ook de attitude of houding van gebruikers ten opzichte van het gebruik van de software. Onder gebruikers valt elk type gebruiker variërend van opdrachtgevers, gebruikersmanagement en onderhoudspersoneel.

Uiteraard dient er een relatie te bestaan tussen kwaliteit-in-gebruik, de vier kenmerken daarvan en de zes eerder genoemde hoofdkenmerken van softwarekwaliteit. ISO/IEC stelt dat kwaliteit-in-gebruik in feite de resultante is van de manier waarop en de mate waarin de zes hoofdkwaliteitskenmerken zijn gerealiseerd. Daarbij wordt opgemerkt dat het doel van het bewerkstelligen van softwarekwaliteit niet 'perfecte kwaliteit' is, maar noodzakelijke en voldoende kwaliteit voor de specifieke gebruiksomgeving waarvoor de software wordt ontwikkeld.

Voor wat betreft het bepalen van kwaliteit-in-gebruik, in termen van de vier kenmerken, wordt opgemerkt dat expliciet gemaakte kwaliteitsbehoeften van gebruikers niet altijd hun werkelijke behoeften weergeven. Redenen daarvan zijn onder meer dat:

- gebruikers zich niet altijd bewust zijn van hun werkelijke behoeften
- gebruikersbehoeften kunnen veranderen nadat ze zijn geformuleerd
- het praktisch gezien onmogelijk is om alle taakgebieden, werkprocessen en gebruiksmogelijkheden diepgaand te analyseren.

Ook zal het niet altijd mogelijk zijn om kwaliteitseisen voorafgaand aan het eigenlijke ontwikkelwerk volledig te specificeren in termen van de zes hoofdkwaliteitskenmerken. In dat geval dienen de specificaties van de kwaliteitseisen tijdens ontwikkeling, bijvoorbeeld op basis van prototyping, te worden bijgesteld of aangepast.

Zoals tussen kwaliteit-in-gebruik en kwaliteit vanuit het externe perspectief relaties bestaan, zo bestaan ook tussen het externe en interne perspectief van softwarekwaliteit relaties. Aan de hand van de interne en externe metrieken kan worden getracht deze onderlinge relaties aan te tonen.

ISO/IEC stelt dat proceskwaliteit, ofwel de kwaliteit van ontwikkelprocessen, bijdraagt aan het verbeteren van softwarekwaliteit, en dat de eenmaal gerealiseerde softwarekwaliteit bijdraagt aan kwaliteit-in-gebruik. Daarom is ook het evalueren en verbeteren van processen een middel om softwarekwaliteit te verbeteren, evenals dat het evalueren en verbeteren van software kwaliteit een middel is om kwaliteit-in-gebruik te verbeteren. Omgekeerd kan het evalueren van kwaliteit-in-gebruik leiden tot het aanbrengen van verbeteringen in de zes hoofdkenmerken van softwarekwaliteit en kan het evalueren van de zes hoofdkenmerken leiden tot het aanbrengen van verbeteringen in ontwikkelprocessen.

Opgemerkt dient te worden dat de kwaliteit van de informatie, die wordt gegenereerd door software, niet in de ISO 9126 standaard wordt behandeld. We geven daarom hieronder een korte toelichting op dit onderwerp

6.3.3 Kwaliteit van informatie

Over de kwaliteit van informatie zijn de afgelopen jaren verschillende belangrijke artikelen verschenen. Strong e.a. (1997) laat zien dat meer en meer bedrijven de kwaliteit van informatie wensen te benoemen en te beschrijven. De gebrekkige kwaliteit van data en de informatie die wordt gegenereerd door software, veroorzaakt schadeposten van miljarden dollars (Wang, 1995).

Het artikel (Strong e.a., 1997) stelt dat aandacht voor de zogeheten intrinsieke kwaliteit van data, zoals gespecificeerd en geïmplementeerd in software, niet voldoende is om gebruikers of organisaties tevreden te stellen en/of afdoende te ondersteunen. Kwaliteit van informatie (ook wel extrinsieke kwaliteit genoemd) dient te worden benaderd vanuit de omgevingsbehoeften en/of –organisatiekenmerken van de software. Daarbij gaat het bijvoorbeeld om de behoeften van gebruikers (zoals toegankelijkheid van data) en/of vanuit een organisatiestandpunt (zoals beveiliging).

In het genoemde artikel wordt, om kwaliteit van data bespreekbaar en hanteerbaar te maken, onderscheid gemaakt in kwaliteitscategorieën. De categorieën die worden onderscheiden zijn respectievelijk:

- *intrinsieke kwaliteit*, bedoeld worden accuraatheid, objectiviteit, geloofwaardigheid en reputatie van de data
- *toegankelijkheidskwaliteit*, genoemd wordt onder meer de beveiliging van de data
- *contextuele kwaliteit*, hier worden genoemd de relevantie, de toegevoegde waarde, de tijdigheid, de volledigheid en de hoeveelheid data
- *representatiekwaliteit*, genoemd worden de interpreteerbaarheid, het gemak waarmee data kunnen worden begrepen, aantrekkelijkheid en consistentie van de representatie.

Op basis van deze indelingen en begrippen zijn problemen op het vlak van informatiekwaliteit in verschillende bedrijfssituaties onderzocht. Casestudies tonen aan dat een dergelijk raamwerk voor datakwaliteit noodzakelijk is. Aangetoond werd bijvoorbeeld dat het hanteren van dit taalgebruik leidt tot het verrijken van het inzicht binnen organisaties voor wat betreft het stellen van expliciete eisen aan datakwaliteit. Ook werd duidelijk dat het hanteren van de categorieën van datakwaliteitskenmerken kan leiden tot aanvullende eisen die dienen te worden gesteld aan de software, bijvoorbeeld voor wat betreft in te bouwen beveiligingsmechanismen.

6.4 Verschillende beschouwingswijzen van software: verschillende kwaliteitskenmerken

Kwaliteit-in-gebruik van software wordt bepaald door de manier waarop gebruikers de software gebruiken binnen een bepaalde bedrijf- of productomgeving.

Dit houdt in dat de kenmerken van een bedrijf- of productomgeving het belang bepalen van de kwaliteit-in-gebruik kenmerken. Bijvoorbeeld in geval binnen een bedrijfsomgeving belangrijke managementbeslissingen moeten worden genomen op strategisch niveau en daarvoor gebruik wordt gemaakt van een informatiesysteem dan zal het kwaliteit-in-gebruik-kenmerk 'effectiviteit' van groot belang zijn. Vraagt een bedrijfsomgeving om een efficiënte ondersteuning van routinematig werk dan is met name het kwaliteit-in-gebruik-kenmerk 'productiviteit' van belang. De stelling 'verschillende bedrijfsomgevingen: verschillende kwaliteit-in-gebruik-kenmerken' kan worden uitgebreid met de stelling 'verschillende beschouwingswijzen, verschillende kwaliteitskenmerken'. In de dissertatie van Trienekens (1994) worden drie invalshoeken gehanteerd van waaruit software kan worden bezien. In het navolgende wordt met voorbeelden aangetoond dat afhankelijk van de manier waarop software wordt beschouwd, verschillende kwaliteitskenmerken al dan niet terecht worden benadrukt. De drie invalshoeken zijn respectievelijk:
- de productbegrenzing van software
- de fasen die software doorloopt
- de uniekheid van software.

Met name binnen de eerste invalshoek worden kwaliteitskenmerken van software vaak onterecht over het hoofd gezien. We zullen daarop kort ingaan in de navolgende paragraaf.

6.4.1 Kwaliteitskenmerken variëren met de productbegrenzing van software

Software is vaak niet eenduidig afgebakend en gedefinieerd. Afwisselend wordt er onder verstaan: de programmatuur (of sourcecode), een 'softwarepakket' inclusief handleidingen en procedures, maar soms wordt ook de 'output', ofwel de informatie die wordt gegenereerd

door de software, tot die software gerekend (Bemelmans 1998). Kennelijk wisselt in de praktijk de productbegrenzing van software.

We beschrijven in deze paragraaf twee verschillende productbegrenzingen van software.
1. de '*enge*' begrenzing van software als programmatuur
2. de '*ruime*' productbegrenzing die is uitgebreid tot en met de directe omgeving van de software.

De consequenties van het hanteren van verschillende productbegrenzingen bij het benoemen van kwaliteitskenmerken worden met enkele voorbeelden toegelicht. Opgemerkt dient te worden dat het eigenlijk principieel onjuist is dat software in de 'enge' productbegrenzing wordt ontwikkeld en wordt verkocht aan een opdrachtgever. De 'enge' productbegrenzing houdt namelijk in dat men bepaalde kwaliteitskenmerken, die zijn gerelateerd aan de gebruiksomgeving, en daarom altijd en overal in een bepaalde mate relevant zijn voor gebruikers, min of meer bewust over het hoofd ziet. We komen daarop terug in het navolgende.

Software met een 'enge' productbegrenzing ofwel Software 'an sich'
In het model voor softwarekwaliteit van Boehm c.s. (1976) heeft kwaliteit betrekking op software 'an sich', zie figuur 6.1. Boehm geeft expliciet aan dat daarmee de source-code wordt bedoeld. Door Willmer (1985) wordt ook documentatie tot de software gerekend, zowel de handleidingen voor de gebruiker als voor de beheerder. Specifieke kwaliteitskenmerken daarvan worden echter niet gegeven. We bespreken kort op welke manier in deze situatie kwaliteit wordt benaderd en gehanteerd.

Kwaliteit van software met een 'enge' productbegrenzing
Software engineers trachten kwaliteit te realiseren door de softwareprogrammatuur gestructureerd te ontwerpen, de efficiency van de programmatuur te vergroten, het computergeheugengebruik te verbeteren, de toegankelijkheid van gegevensbestanden te verbeteren, etc. Doel is om door middel van het nemen van technische (ontwerp)maatregelen kwaliteitskenmerken te realiseren zoals responsesnelheid, overdraagbaarheid en koppelbaarheid. Voor de realisatie van die eisen wordt onder meer gebruik gemaakt van technische standaards. Bij deze werkzaamheden wordt *geen* rekening gehouden met specifieke kenmerken van een gebruiksomgevings waarbinnen de software moet functioneren en presteren.

Software met een ruime productbegrenzing
In de kwaliteitsopvatting van Bemelmans (1998) is software niet beperkt tot programmatuur alleen, al dan niet voorzien van documentatie. Software dient, werkend binnen een bedrijfsomgeving, te worden beschouwd als een informatiesysteem. Tevens dient de 'output', ofwel de informatie, die door de software wordt gegenereerd daarbij te worden betrokken. De productbegrenzing van software wordt daarmee aanzienlijk uitgebreid. Immers zowel de software, de integriteit van de data die wordt gegenereerd, de bijbehorende documentatie, maar ook de procedures voor de aansluiting op de taken van gebruikers vallen nu onder de definitie van software.

De kwaliteit van software met een ruime productbegrenzing is afhankelijk van meer dan alleen de kwaliteit van de programmatuur. in een praktijkonderzoek naar de waardering van gebruikers voor software wordt dit beschreven, zie o.a Bailey en Pearson (1983). kwaliteit heeft volgens kim (1990) dan te maken met:

- de procedures voor invoer en de uitvoer van de software
- de efficiency van de gegevensverwerking in de software
- de ondersteuning die de software biedt
- de effectiviteit van de informatie die de software genereert.

We lichten kwaliteit van software met een ruime productbegrenzing in het navolgende toe met enkele voorbeelden van kwaliteitseisen.

Kwaliteit van software met een ruime productbegrenzing
Om kwaliteitseisen van software ingepast in een bedrijfsomgeving te kunnen beschrijven, dient die omgeving te worden onderzocht. Kwaliteitseisen die dan kunnen worden afgeleid zijn bijvoorbeeld 'de aansluiting van de software op de handmatige procedures' binnen een bepaalde bedrijfsfunctie of taakgebied. Of: de gewenste mate van ondersteuning van taken binnen een betreffende bedrijfsfunctie, of de beschikbaarheid naar tijd en/of plaats van software. Voor het afleiden van dit soort kwaliteitseisen is specifieke kennis van het taakgebied en van de gebruikers daarbinnen noodzakelijk. Nog belangrijker is kennis en inzicht in een bedrijfssituatie wanneer kwaliteitseisen van de informatie, die door de software wordt gegenereerd, moet worden gespecificeerd. Deze kwaliteitseisen betreffen dan bijvoorbeeld de effectiviteit van de gegenereerde informatie voor bepaalde besluitvormingsprocessen.

Bovenstaande maakt duidelijk dat softwarekwaliteit varieert met de productbegrenzing die wordt gehanteerd. we merken nogmaals op dat een 'enge' productbegrenzing van software 'an sich', ofwel software die zich louter beperkt tot de programmatuur, in feite nooit te rechvaardigen is. daardoor wordt het omgaan met softwarekwaliteit beperkt tot een uitsluitend technisch karwei dat ver verwijderd blijft van de kwaliteit die door een gebruiksomgeving wordt gewenst of vereist.

6.4.2 Kwaliteitskenmerken variëren met de levenscyclusfasen van software

In diverse publicaties komt de kwaliteit van tussenproducten van softwareontwikkeling aan de orde, zie bijvoorbeeld Basili en Rombach (1988), Bush en Fenton (1990). Gesteld wordt onder meer dat kwaliteitsbeheersing dient te zijn gebaseerd op het per deelproduct of tussenproduct verrichten van metingen. Kwaliteitseisen en/of kenmerken kunnen sterk variëren per fase van het ontwikkelproces of per toestand waarin de software zich bevindt.

Oplevering van de software, bijvoorbeeld overdracht aan de klant of gebruiker, betekent niet per definitie het einde van het ontwikkelwerk. Door Looyen (1993) worden verschillende toestanden beschreven waarin de software zich kan bevinden. Respectievelijk wordt onderscheid gemaakt tussen de toestand:
1. vóór oplevering van software (de ontwikkeltoestand)
2. van invoering in de organisatie (ná oplevering)
3. van beheer en exploitatie
4. van wijziging.

Met betrekking tot de laatste toestand wordt opgemerkt dat in geval van wijziging, waaronder ook uitbreiding en/of verandering valt, de software (tijdelijk) weer in een ontwikkeltoestand verkeert.

We bespreken in het navolgende twee toestanden van de software, die we respectievelijk zullen beschrijven als software tijdens projectmanagement en software tijdens productmanagement (van Genuchten 1990). Met software tijdens projectmanagement bedoelen we software die projectmatig tot stand komt. Bij het afsluiten van het project wordt de software als het ware 'over de muur gegooid', hetgeen wil zeggen dat ontwikkelaar er verder in het geheel geen verantwoordelijkheid meer voor draagt. Deze toestand komt dus overeen met de zogenoemde 'ontwikkeltoestand'. Voor software tijdens productmanagement ligt dit anders. Ontwikkelaar en opdrachtgever dragen gedurende de gehele levenscylcus een gezamenlijke verantwoordelijkheid voor de software. In feite is deze situatie een optelsom van alle vier genoemde toestanden.

Toestanden van software tijdens projectmanagement
Projectmanagement richt zich op de beheersing van het ontwikkelwerk vanaf het vastleggen van eisen tot en met de constructie van software. Volgens Basili en Rombach (1988) en (Bush en Fenton (1990) kunnen op alle deel- en tussenproducten metingen worden verricht. In overleg met projectleiders en ontwikkelaars dient afhankelijk van de situatie afgesproken te worden welke kwaliteitseisen relevant zijn en welke metrieken dienen te worden gehanteerd. Voorbeelden van tussenproducten zijn: conceptuele gegevensmodellen, programmatuur, databasestructuren etc. Een voorbeeld van een kwaliteitseis is de gestructureerdheid van de software.

Doel van projectmanagement is het opleveren van software overeenkomstig de specificaties en binnen de afgesproken tijd en kosten. Zoals eerder vermeld, weten ontwikkelaars zich bij oplevering van de software ontslagen van verantwoordelijkheden. Invoering, exploitatie, onderhoud en verandering, komen geheel en al voor rekening van de gebruiker of de opdrachtgever.

Kwaliteit van software tijdens projectmanagement
De kwaliteit van software verandert tijdens het ontwikkelwerk en naar we mogen aannemen betekent zo'n verandering een verbetering. Ontwikkelaars trachten de specificaties zo goed mogelijk te vertalen in kenmerken van de software. Tussentijds vinden verificaties en validaties plaats van analyse- en ontwerpactiviteiten. Afhankelijk van de intensiteit waarmee opdrachtgevers en/of gebruikers participeren in het ontwikkelwerk wordt door hen mede de kwaliteit bepaald. Aan kwaliteitseisen zoals gebruiksvriendelijkheid, bijvoorbeeld de inzichtelijkheid van de schermlay-out, kan dan aandacht worden besteed. Omdat ontwikkelaars na oplevering geen verantwoordelijkheden meer dragen, hebben zij er geen belang bij om al teveel tijd te besteden aan zaken als de onderhoudbaarheid, de flexibiliteit of de portabiliteit van software, dus kwaliteitseisen die een rol spelen na oplevering van de software. Deze eigenschappen zullen dan ook hooguit nominaal zijn, dat wil zeggen in overeenstemming met algemeen geldende standaards zoals die bijvoorbeeld gelden voor een programmeertaal of een hardwareplatform. Ook aan kwaliteitskenmerken zoals de inleertijd van de software behoeft eigenlijk door ontwikkelaars nauwelijks of geen aandacht te worden besteed.

Toestanden van software tijdens productmanagement
Gebruikers eisen in toenemende mate dat ontwikkelaars betrokken blijven bij software in de toestanden ná oplevering. Softwareontwikkeling is niet langer beperkt tot een 'ontwerpcyclus, maar strekt zich uit over de gehele levenscyclus van de software, inclusief onderhoud, wijziging en uitbreiding. Dit wordt *softwareproductmanagement* genoemd. Zoals eerder vermeld, is het belangrijkste verschil ten opzichte van projectmanagement de doorlopende

gedeelde verantwoordelijkheid van ontwikkelaars en gebruikers voor het functioneren en de kwaliteit van software, zowel tijdens het ontwerpwerk als tijdens beheer (o.a. wijziging) en exploitatie (o.a. invoering).

Kwaliteit van software tijdens productmanagement
Softwareproductmanagement omvat de toestanden van software voor en na overdracht aan de gebruiker. Dit betekent dat ontwikkelaars en gebruikers een gezamenlijk belang hebben bij het bewerkstelligen van onderhoudbaarheid, flexibiliteit en transitiemogelijkheden van de software. Softwarekwaliteit wordt voor gebruikers afhankelijk van bijvoorbeeld de geboden begeleiding bij de invoering van de software, de overeenkomsten op serviceniveau, de afspraken over nieuwe versies van de software, de integratiemogelijkheden etc. Al deze aspecten kunnen tijdens productmanagement leiden tot specifieke eisen aan de kwaliteit van de software. Een voorbeeld van een kwaliteitseis tijdens exploitatie is onderhoudbaarheid en voorbeelden tijdens transitie zijn aanpasbaarheid, converteerbaarheid, en compatibiliteit.

Softwarekwaliteit varieert met de toestanden, ofwel de fasen, die de software tijdens de levenscyclus doorloopt. het bovenstaande maakt duidelijk dat afhankelijk van de fasen die men wenst mee te nemen in de beschouwing verschillende kwaliteitskenmerken worden benadrukt. of dit terecht is voor wat betreft het uiteindelijke gebruik van de software hangt af van de afspraken die ontwikkelaar en opdrachtgever onderling hebben gemaakt over de verantwoordelijkheden voor bijvoorbeeld de invoering, het onderhoud en de verbetering van de software.

6.4.3 Kwaliteitskenmerken variëren met de mate van uniekheid van software

Veel ontwikkelaars en opdrachtgevers of gebruikers kunnen het zich niet langer permitteren om voor elke bedrijfssituatie specifieke 'maatwerksoftware te laten ontwikkelen. Voor ontwikkelaars is dit een gevolg van de toenemende concurrentie binnen de software-industrie. Het sneller kunnen voldoen aan specifieke wensen van klanten wordt een halszaak. Aanleidingen voor gebruikers zijn enerzijds de toename in tijd en kosten die maatwerkoplossingen met zich meebrengen en anderzijds de beperkte mogelijkheden van uitbreiding en aanpassing van dit soort software. Meer en meer wint de opvatting terrein dat hergebruik van software(componenten) een oplossing is voor genoemde problemen. In deel III van dit boek wordt nader ingegaan op het vraagstuk 'make or buy' en de overwegingen die een rol spelen om software zelf te (laten) maken of (standaard)software te kopen. Hergebruik komt in deel IV nader aan de orde als een middel om de kwaliteit van software te beïnvloeden. Doel van hergebruik is om met een gelimiteerd aantal basiscomponenten software samen te stellen waarmee tegemoet wordt gekomen aan de wensen van een klant. Afhankelijk van de standaardcomponenten die worden gebruikt is software meer of minder uniek. Uiteraard kan samengestelde software wel uniek zijn op grond van de specifieke wijze van configuratie uit standaardcomponenten.

We bespreken in het navolgende twee uitersten met betrekking tot de uniekheid van software, te weten:
- maatwerksoftware
- standaardsoftware.

Twee begrippen staan daarbij centraal: het begrip klantafhankelijkheid en het begrip hergebruik.

Maatwerksoftware
Met maatwerksoftware wordt bedoeld software waarbij men tijdens de ontwikkeling heeft getracht zo volledig mogelijk rekening te houden met de specifieke informatiebehoeften van een bedrijfssituatie. Maatwerksoftware is dus in het uiterste geval uniek. De ontwikkeling van dergelijke software is sterk klantafhankelijk *(de klant specificeert!)*. Dit betekent dat gebruikers intensief betrokken zijn bij zowel het definiëren en specificeren van eisen als bij de verificatie en validatie van tussen- en eindproducten. Voor ontwikkelaars ontbreken immers andere soorten referentiemateriaal. Hergebruik is in deze situatie nauwelijks mogelijk. Uitgangspunt is kennis en ervaring in de hoofden van ontwikkelaars en opdrachtgevers of klanten. Dit type softwareontwikkeling staat ook bekend als het verkopen van puur productie- of ontwikkelcapaciteit door ontwikkelaars aan opdrachtgevers (Trienekens en Kusters, 1992). Ontwikkelaars zijn niet gespecialiseerd in bepaalde soorten software. Door ontwikkelaars wordt nauwelijks met ontwikkelstandaards gewerkt. Men tracht zich volledig te richten naar de wensen en de voorkeuren van gebruikers.

In minder extreme vormen van de maatwerkontwikkeling worden referentiemodellen gebruikt. Referentiemodellen weerspiegelen eerder gedaan werk op het gebied van het structureren van de bedrijfsvoering en de informatievoorziening (Greveling, 1990).

Kwaliteit van maatwerksoftware
Het criterium voor kwaliteit is tevredenheid bij de klant. functionele en technische onvolkomenheden kunnen acceptabel zijn voor ontwikkelaars mits de tevredenheid van de opdrachtgever daar niet onder lijdt. naadloze inpassing van software binnen bedrijfsprocessen zal bijvoorbeeld belangrijker zijn dan hoge eisen aan modulariteit en uitbreidbaarheid van de software.

Ontwikkelaars richten zich sterk op het aftasten van onuitgesproken wensen én behoeften van klanten. Zowel objectieve als subjectieve eisen en wensen tracht men mee te nemen als uitgangspunt bij het ontwerp. Door allerlei, eventueel niet gespecificeerde, extra's kan worden getracht de kwaliteit te verhogen. Dit kan variëren van het zo volledig mogelijk aansluiten van de software op handmatige procedures binnen een bedrijfssituatie tot het vergroten van de aantrekkelijkheid van de software door het aanbrengen van 'toeters en bellen', bijvoorbeeld door extra aandacht te besteden aan kleurgebruik, geluid en vormaspecten.

Standaardsoftware
Standaardsoftware kan in meerdere soortgelijke bedrijfssituaties, bijvoorbeeld bedrijfsbranches, worden toegepast. Ontwikkelaars van standaardsoftware beperken zich over het algemeen tot bepaalde 'productfamilies'. Ontwikkelaars zijn gericht op het ontwikkelen van softwarecomponenten met het oog op hergebruik. De componenten worden in plaats van klantafhankelijk, marktafhankelijk ontwikkeld. Het ontwikkelwerk wordt zoveel mogelijk gestandaardiseerd bijvoorbeeld qua structurering en codering. Van herbruikbaar materiaal bestaan inmiddels vele definities. Bijvoorbeeld door Trienekens en Kusters (1992) wordt onderscheid gemaakt in hergebruik van informele componenten, zoals bijvoorbeeld referentiemodellen, en in hergebruik van formele componenten, zoals bijvoorbeeld programmaprocedures, database-structuren, en mensmachinedialogen. Ook wordt onderscheid gemaakt in herbruikbare product- en procescomponenten. Voorbeelden van procescomponenten zijn: levenscyclusmodellen (bijvoorbeeld: lineair, incrementeel, prototyping) maar ook gedetailleerde activiteitenstructuren (bijvoorbeeld voor het normaliseren van gegevensstructuren).

Doel van hergebruik is om op efficiënte wijze op basis van een gelimiteerd aantal componenten of bouwstenen varianten binnen voorgedefinieerde productfamilies te ontwikkelen. Meerdere softwareproducten worden parallel aan elkaar ontwikkeld, zoveel als mogelijk gebaseerd op dezelfde basiscomponenten.

Er bestaan verschillende vormen van standaardsoftware. Bijvoorbeeld een softwarepakket dat parameteriseerbaar is binnen een bepaalde bedrijfssituatie voor een bepaald soort productiebesturing. Een ander voorbeeld is een standaardpakket dat als infrastructurele component organisatiebreed wordt verspreid, bijvoorbeeld een tekstverwerkingspakket, een spreadsheetapplicatie etc.).

Kwaliteit van standaardsoftware
Formeel hergebruik van componenten bij de ontwikkeling van standaardsoftware is slechts mogelijk wanneer bouwstenen met het oog op hergebruik worden ontwikkeld. Dit betekent dat deze bouwstenen een hoge graad van perfectie dienen te hebben. Zwakke plekken in componenten kunnen immers in een (nieuwe) configuraties fataal zijn. Standaardisatie en modularisatie van bouwstenen zijn belangrijk kwaliteitseisen voor deel- of tussenproducten van het ontwikkelwerk. Deze kenmerken vormen een voorwaarde voor het voldoen van eisen zoals flexibiliteit, koppelbaarheid en overdraagbaarheid.

Het voorgaande toont aan dat softwarekwaliteit varieert met de mate van uniekheid van de software. opdrachtgevers dienen zich te realiseren dat men in geval van maatwerksoftware in feite in staat moet zijn de gewenste en vereiste kwaliteitskenmerken zelf te identificeren en te specificeren. in geval van standaardsoftware is men, in elk geval gedeeltelijk, afhankelijk van de visie, de doelstellingen en de vaardigheden van de ontwikkelaar.

In deze paragraaf is beschreven dat software op verschillende manieren kan worden beschouwd. De drie invalshoeken die we hebben gehanteerd zijn respectievelijk:
- de productbegrenzing van software
- de toestanden die software, tijdens de levenscyclus doorloopt
- de uniekheid van software.

Met voorbeelden is toegelicht dat binnen elke invalshoek de definitie van softwarekwaliteit verschuift en dat daarmee verschillende kwaliteitskenmerken naar voren komen.

6.5 Kwaliteitskenmerken zijn afhankelijk van actoren

Ook in het boek van Van Zeist e.a. (1996) wordt nader ingegaan op het thema 'verschillende situaties, verschillende kwaliteitskenmerken'. Van Zeist e.a. gaan echter uit van de gezichtspunten van verschillende typen personen die zijn betrokken bij software(-ontwikkeling) en trachten vanuit de karakteristieken van die personen aan te geven welke kwaliteitskenmerken meer of minder relevant zijn.

De typen personen die worden onderscheiden zijn respectievelijk:
- *de gebruiker*, onderverdeeld in respectievelijk de eindgebruiker, de koper (gebruikersmanager) en de gebruikersondersteuner
- *de ontwikkelaar*, onderverdeeld in productontwerper, productontwikkelaar, product-beheerder en projectmanager
- *de evaluator*, onderverdeeld in de productevaluator, de certificeerder en de EDP-auditor.

We lichten dit toe aan de hand van enkele voorbeelden

> NB: voor de Engelstalige definities van kwaliteitskenmerken zie de betreffende Appendix 2. De begrippen die in deze appendix worden gepresenteerd, worden in de oorspronkelijke Engelstalige omschrijving gegeven. Dit is bewust gedaan omdat de begrippen na veel wikken en wegen door de ISO/IEC uiterst zorgvuldig zijn gedefinieerd. Bij een vertaling naar het Nederlands zou een deel van deze zorgvuldigheid verloren kunnen gaan..

Eindgebruiker
Vanuit het gezichtspunt van de eindgebruiker moet het softwareproduct zijn dagelijkse werk ondersteunen. Het moet de gebruiker helpen bij het uitvoeren van zijn taken en moet die vereenvoudigen. Kleine ongemakken kunnen resulteren in grote ergernis. De consequentie van dit gezichtspunt voor de inhoud van een specificatie van kwaliteitskenmerken is dat over het algemeen een sterke nadruk ligt op usability-kenmerken, bijvoorbeeld het subkenmerk attractiveness. Daarnaast zijn efficiency kenmerken, met name time-behaviour, van belang. Binnen het kwaliteitskenmerk functionality zal vaker het subkenmerk suitability hoog op de ranglijst staan.

Koper (gebruikersmanager)
Een koper zal willen beschikken over een duidelijke en volledige kwaliteitsspecificatie. Een belangrijk doel van een koper, in een rol van gebruikersmanager, is tevredenheid en instemming onder de gebruikers. De aandacht vanuit dit gezichtspunt gaat meestal in de richting van functionality- en efficiency-kenmerken. Door een manager kan bijvoorbeeld een risico-analyse worden gedaan om het belang van bepaalde kenmerken te bepalen. Vaak zijn portability-kenmerken minder van belang, tenzij de manager verantwoordelijk is voor gebruikers die op heterogene systeemplatformen werken.

Gebruikersondersteuner
Dit gezichtspunt treffen we aan bij personen die gebruikers assisteren met het werken met software. Wijzigen, aanpassen aan wensen en installeren, maken onderdeel uit van de primaire taken onderhoud en ondersteuning. Binnen gebruikersondersteuning wordt vaak veel nadruk gelegd op portability-kwaliteitskenmerken zoals installability en adaptability, en maintainability-eigenschappen zoals stability en manageability.

Ontwikkelaar (engineer, ontwerper, projectmanager, beheerder)
Een ontwikkelaar is verantwoordelijk voor het verwerken van alle software-eisen in het software-ontwerp. Vanuit dit gezichtspunt bestaat grote behoefte aan duidelijke en praktisch hanteerbare maatregelen die kunnen worden genomen om kwaliteitskenmerken in te bouwen in de software. Daarnaast zijn metrieken noodzakelijk om te kunnen vaststellen of bepaalde maatregelen tot de gewenste effecten hebben geleid. Voor de productbeheerder wordt wel expliciet een belangrijk kwaliteitskenmerk genoemd, te weten maintainability, met name analysability en traceability.

Evaluator
De gezichtspunten van de evaluator zijn gebaseerd op verschillende soorten productevaluaties. Respectievelijk worden onderscheiden:
- De algemene productevaluatie, een product wordt geëvalueerd om het te kunnen vergelijken met andere producten. Praktijkonderzoek wijst uit dat de algemene productevaluator

zich in de praktijk voornamelijk concentreert op functionality, met bijzondere aandacht voor 'geschiktheid voor gebruik' ofwel suitability.
- Een certificatie, een product wordt gecertificeerd om de producent of de kopers vertrouwen te geven in de kwaliteit. Voor een certificeerder zijn met name kwaliteitskenmerken zoals conformance en compliance van belang.
- Een EDP-audit, een evaluatie door een EDP-auditor wordt ook wel systeem-audit genoemd. Een EDP-audit kan worden verricht op verzoek van een accountant als ondersteuning van diens accountantscontrole. Voor een EDP-auditor staat het kwaliteitskenmerk reliability centraal. De controleerbaarheid en de beheersbaarheid van bijvoorbeeld verwerkingsprocessen binnen of door de software zijn een belangrijk aandachtspunt. Ook beschikbaarheid van (informatie die wordt gegenereerd door) de software is een relevant kwaliteitskenmerk.

Bovenstaande laat zien dat vanuit verschillende gezichtspunten verschillende kwaliteitskenmerken op de voorgrond kunnen worden gesteld. In discussie over gewenste en/of noodzakelijke kwaliteit is het daarom raadzaam eerst duidelijk te krijgen met welke betrokken personen men die discussie voert, en wat hun belangen en hun taken zijn. Op grond daarvan kan wellicht op een meer efficiënte en effectieve wijze een specificatie van kwaliteitskenmerken tot stand komen.

6.6 Conclusies

In dit hoofdstuk hebben we ons gericht op een nadere beschrijving van het begrip softwarekwaliteit. Nadat we een aantal bekende opvattingen en de verschillen en overeenkomsten tussen deze opvattingen de revue hebben laten passeren, hebben we de meest recente 'state of the art' definitie van softwarekwaliteit behandeld, namelijk de definitie volgens ISO/IEC 9126. Deze standaard, die weliswaar nog in ontwikkeling is, laat met via het begrip 'kwaliteit-in-gebruik' zien dat kwaliteit *kan* of beter *moet* worden afgeleid uit de bedrijfs- en/of gebruiksomgeving waar software wordt toegepast. Door ISO/IEC wordt een contingentieprincipe naar voren gebracht: dat wil zeggen dat verschillende situaties vragen om verschillende kwaliteitskenmerken.

In paragraaf 6.4 hebben we, dit principe indachtig, uitgelegd dat afhankelijk van de beschouwingswijze van software, in een specifieke situatie verschillende kwaliteitskenmerken een rol spelen. Een en ander maakt duidelijk dat softwarekwaliteit en daarmee kwaliteitskenmerken moeten worden afgeleid uit de bedrijfs- of gebruiksomgeving in kwestie. In hoofdstuk 7 wordt deze opvatting verder uitgewerkt en worden benaderingen beschreven voor het afleiden van kwaliteitskenmerken uit de behoeften van gebruikers in een bepaalde bedrijfsomgevingen.

Artikel 15 bij leereenheid 5

Het realiseren, handhaven en garanderen van een goed softwareproces

Uit: *Softwarekwaliteit*
Fred. J. Heemstra, Rob. J. Kusters en Jos J.M. Trienekens (2002), pp. 198-203

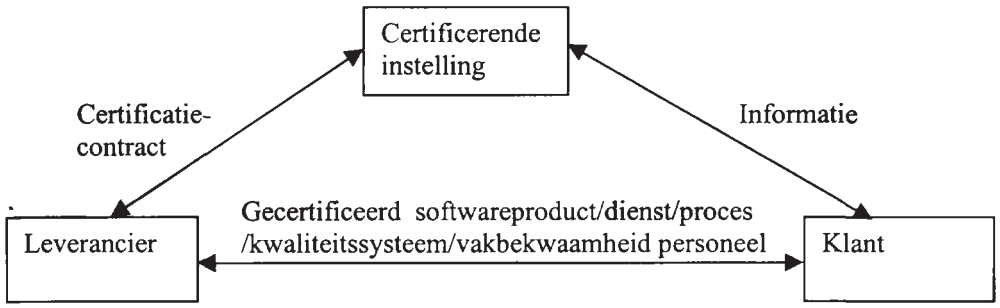
18.4 Garanderen van proceskwaliteit: certificeren

18.4.1 Inleiding tot certificeren

Een klant (intern dan wel extern), die software laat ontwikkelen of inkoopt van een leverancier, zal als eis stellen dat voldaan wordt aan de geformuleerde kwaliteitseisen. Het komt meer dan eens voor dat de klant niet in staat of niet bereid is zelf te beoordelen of hieraan inderdaad is voldaan. In zo'n geval kan gebruik worden gemaakt van certificatie. Waszink (Waszink, 1991) omschrijft in zijn inaugurale rede certificatie als volgt:

> *'Certificatie betekent dat wordt voorzien in een onafhankelijke bewijslast op het niveau van bedrijven en instellingen. Het omvat activiteiten op grond waarvan een onafhankelijke instantie kenbaar maakt dat een gerechtvaardigd vertrouwen bestaat, dat een duidelijk omschreven onderwerp van certificatie in overeenstemming is met een bepaalde norm of een ander normstellend document'.*

Eén en ander impliceert dat we bij certificatie te maken hebben met een derde partij i.c. de certificerende instantie. In figuur 18.3 wordt de positie van een dergelijke instantie ten opzichte van klant en leverancier in beeld gebracht.



Figuur 18.3: Relatie tussen klant, leverancier en certificerende instelling (Waszink, 1991)

Bij certificatie is het belangrijk dat betrokken partijen overeenstemming hebben over datgene wat gecertificeerd dient te worden (welke eigenschappen op basis van welke eisen), hoe gecertificeerd wordt (de gehanteerde certificatiemethode) en de omstandigheden waaronder de afgegeven certificatie van toepassing is.

De onderwerpen die voor certificatie in aanmerking komen zijn producten, diensten, processen, kwaliteitssystemen en de vakbekwaamheid van personeel. In deel II is ingegaan op certificatie van producten en in deel IV zal worden stilgestaan bij vormen van certificatie van de vakbekwaamheid. In dit procesdeel richten we ons op procescertificatie.

Procescertificatie (Waszink, 1991) bestaat uit een proceskeuring en een controlekeuring van onderdelen van het kwaliteitssysteem. Waszink geeft als voorbeeld het scharrelei. Aan de eieren is niet te zien welke soort kip deze gelegd heeft. Door keuring van de leefomstandigheden van de kip en de distributie van de eieren kan door middel van procescertificatie vertrouwen worden verschaft, dat een ei werkelijk een scharrelei is.

Bij software geldt een min of meer identieke redenering. Juist door het abstracte, onzichtbare en alle overige kenmerken van software (zie deel I) is het voor de klant 'van de buitenkant' lastig te beoordelen of de software voldoet aan bepaalde kwaliteitseisen. Pas bij uitvoerig gebruik kan zo'n oordeel gevormd worden. Door het afgeven van een procescertificatie wordt gegarandeerd dat volgens bepaalde standaards is gewerkt en wordt vertrouwen gewekt dat de software die door dit proces wordt gerealiseerd kwalitatief aan de maat is.

18.4.2 Softwareprocescertificatie

In 1993 gaf Wentink (Wentink, 1993) in zijn inauguratie op treffende wijze een van de problemen weer waarmee softwareontwikkelaars worden geconfronteerd:

> *'Steeds meer bedrijven jagen op een ISO 9000 certificaat, maar dit kwaliteitssysteem kent vele beperkingen en tekortkomingen'.*

Door de druk vanuit de markt worden steeds hogere eisen gesteld aan de voorspelbaarheid van de kwaliteit van producten. Om potentiële klanten te overtuigen van hun kennis en kunde, vluchten steeds meer leveranciers naar het overleggen van een certificaat van het kwaliteitssysteem dat ze hanteren. ISO 9000 is dan het sleutelwoord. Het vermoeden bestaat dat veel aanvragers certificatie nastreven uit marketingoverwegingen. Het certificaat wekt namelijk de indruk dat de organisatie altijd hoge kwaliteit levert terwijl feitelijke

verbeteringen van producten en processen daarmee absoluut niet zijn aangetoond (Swinkels e.a., 1995). Swinkels e.a. vervolgen in hun artikel dat men druk bezig is doel en middelen om te draaien. De toets of een organisatie wel of niet goed functioneert dient namelijk primair gerelateerd te worden aan de prestaties in relatie tot de prijs. Wanneer organisaties met een gecertificeerd kwaliteitssysteem slecht scoren wat betreft de kwaliteit van hun producten, de doorlooptijd en de prijs, dan is er iets fundamenteels mis. Omdat een ISO-certificaat geen garantie blijkt te bieden voor een goede prijs-prestatie-verhouding wordt steeds meer kritiek geleverd op het ISO-certificaat en de daaraan ten grondslag liggende normen.

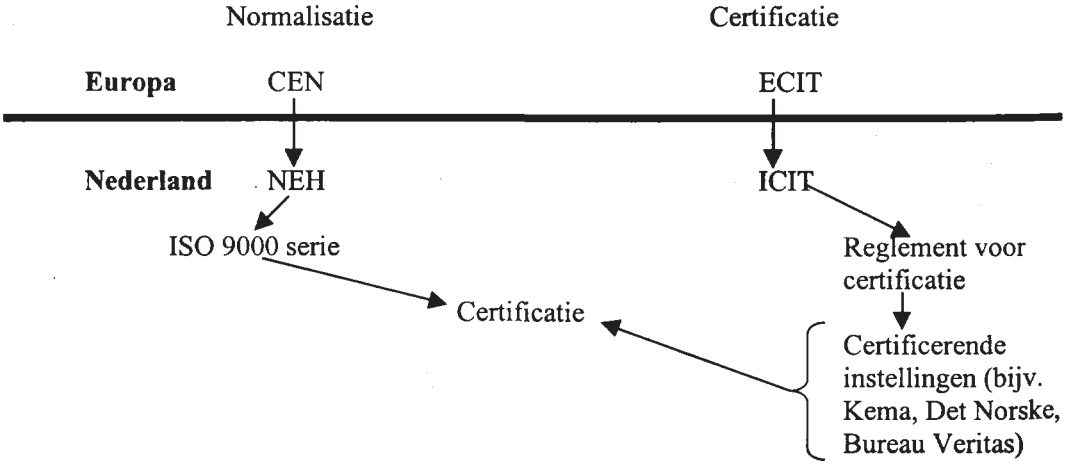
We zullen ons in volgende paragrafen beperken tot certificatie gebaseerd op ISO 9000 en het CMM.

18.4.3 ISO-certificatie

Kan (Kan, 1995) maakt de volgende opmerkingen over ISO-certificatie:

> *'Although many firms and companies are in the process of pursuing ISO 9000 registration, it is interesting to observe that many companies actually fail the audit during the first pass. The number of initial failures ranges from 60% to 70%. This is quite an interesting statistic and is probably explained by the complexity of the standards, their bureaucratic nature, the opportunity for omissions, and a lack of familiarity with what is actually required. From the software standpoint, we observed that corrective action and document control are the areas where most non-conformances were incurred'.*

ISO-certificatie wordt door gespecialiseerde organisaties gedaan. Certificatiebureaus gevestigd in Nederland zijn onder andere de Kema, Det Norske Veritas en bureau Veritas. In het Verenigd Koninkrijk zijn dat ondermeer Lloyd's en het British Standard Insitute en in de USA Underwriter Laboratories. Binnen de Europese Commissie is men druk doende één Europees erkenningsysteem te realiseren door de inrichting van een Europees netwerk (European organization for testing and certification) waaronder in ieder land dergelijke bureaus kunnen worden opgericht. In figuur 18.4 zijn enkele instanties op Nederlands en Europees niveau aangegeven die betrokken zijn bij certificatie.



Figuur 18.4: Certificerende instanties in Nederland en Europa

Het totale pakket aan ISO-documenten beslaat ruim 5.000 pagina's. Het is daarom niet verbazingwekkend dat en directeur van een middelgrote onderneming verzucht (Stevens, 1995):

"We zijn zo druk bezig met certificatie van ons kwaliteitssysteem dat we aan het doorvoeren van verbeteringen niet toekomen".

18.4.4 CMM Certificatie

Binnen het CMM wordt een onderscheid gemaakt tussen 'assessments' en 'evaluations' (Seiadian & Kuzara, 1995). Een assessment van een softwareproces wordt geïnitieerd door een organisatie met als doel haar eigen manier van softwareontwikkeling te verbeteren. In de meeste gevallen wordt zo'n assessment uitgevoerd door zes tot acht seniormedewerkers uit de eigen organisatie en twee coaches, afkomstig van een organisatie die een licentie heeft om CMM-assessments uit te voeren.

Een assessment proces verloopt doorgaans in zes stappen.

1. *In de selectiefase* wordt bepaald of de organisatie geschikt is om een assessment te ondergaan. Een gekwalificeerde extern assessment-bureau (de coaches) rapporteren hierover aan het management
2. *In de commitmentfase* verplicht de organisatie zich te laten onderwerpen aan een assessment en belooft bovendien medewerking op alle gebied. Een en ander wordt bekrachtigd door op het hoogste niveau een zogenoemde assessment agreement te tekenen
3. *In de voorbereidingsfase* wordt het interne assessment team opgeleid en wordt een planning gemaakt voor het assessmentproces. Alle betrokkenen worden ingelicht en de CMM vragenlijsten worden verspreid en ingevuld
4. *In de assessmentfase* wordt gedurende een week de feitelijke assessment uitgevoerd. Hierna komt het assessmentteam bijeen om de voorlopige aanbevelingen te formuleren.
5. *In de rapportagefase* stelt het assessmentteam het eindrapport op en geeft toelichtingen aan alle betrokkenen en het management. Het eindrapport bevat waarnemingen en aanbevelingen voor verbeteringen,
6. *In de assessment-follow-up-fase* wordt door het assessment team, met hulp van de externe coaches een actieplan opgezet. Na ongeveer achttien maanden dient er opnieuw een assessment te worden uitgevoerd om na te gaan of de verbeteringen zijn doorgevoerd en te overzien welke nieuwe verbeteractie opgestart dienen te worden.

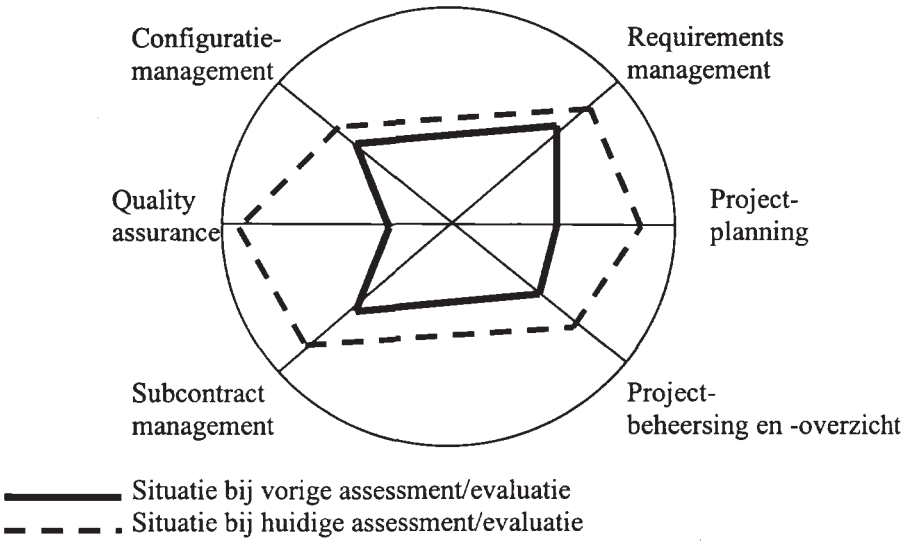
Wil een assessment succesvol zijn, dan vraagt het om veel commitment van de organisatie; een assessment kost tijd en geld zoals we al eerder opmerkten. Betrokkenheid van het hoger management is een voorwaarde voor succes. Een kenmerk van een assessment is bovendien de vertrouwelijkheid van het eindrapport.

In figuur 18.5 wordt ter illustratie een voorbeeld gegeven van een samenvattend overzicht uit een rapportage. Het betreft de wijze waarop de betreffende softwareafdeling scoort op de KPA's voor CMM level 2.

18.4.5 CMM Software Capability Evaluation

Van een heel andere orde is een 'CMM software capability evaluation' (CSE). Van vertrouwelijkheid van het evaluatieresultaat is geen sprake, van vrijwilligheid en eigen

initiatief al evenmin, en bovendien is het niet de eigen organisatie die evalueert maar wordt een CSE uitgevoerd door een extern bureau, vaak een overheidsinstantie of een bedrijf dat veel software inkoopt. Het softwarebedrijf zal zich moeten laten onderwerpen aan een SCE wil het in aanmerking komen om de opdracht voor het ontwikkelen van software voor een klant binnen te halen.



Figuur 18.5: Voorbeeld van een rapportage uit een CMM evaluatie

Een softwareontwikkelorganisatie die in aanmerking wil komen voor een SCE dient eerst de CMM-vragenlijsten in te vullen. Vervolgens bezoekt een evaluatieteam de betreffende organisatie en gaat aan de hand van de ingevulde vragenlijsten bepaalde aspecten nader onderzoeken (bijvoorbeeld de wijze van projectmanagement). Dit nader onderzoek houdt veelal in een uitgebreide interviewronde onder het personeel en een studie van de beschikbare documentatie. Bovendien wordt de wijze van werken van een aantal projecten doorgelicht. Op deze wijze krijgt het evaluatieteam snel een goed inzicht waar potentiële risico's liggen. Een SCE heeft sterk het karakter van een beoordeling, het is niet vrijblijvend. Het hele evaluatieproces is een tijdrovende bezigheid. Het evaluatieteam is weken bezig en legt een grote druk op de organisatie.

In tabel 18.4 wordt de vergelijking tussen een software process assessment en een software capability evaluatie weergegeven.

18.5 Conclusies

In dit hoofdstuk zijn de drie kernvragen ter sprake gekomen:
- hoe dienen de eisen bepaald te worden waaraan een ontwikkelproces moet voldoen?
- hoe wordt een ontwikkelproces gerealiseerd dat voldoet aan deze eisen?
- hoe kan worden gegarandeerd dat wordt voldaan aan deze eisen?

We hebben kunnen zien dat de antwoorden op deze drie vragen nog niet zo uitgekristalliseerd zijn. Er zijn weliswaar modellen als het CMM, ISO, Bootstrap etc. die gebruikt kunnen worden als middel om een steeds beter ontwikkelproces te realiseren. De eisen waaraan het

ontwikkelproces van een specifieke organisatie dient te voldoen, zijn echter niet afkomstig van die eigen organisatie, maar worden min of meer opgelegd door de ISO-eisen of de eisen die door een bepaald CMM level worden gesteld. Zaak is dat een organisatie zelf zoekt naar de proceseisen die het beste aansluiten bij haar eigen ontwikkelproces. De genoemde proceskwaliteitsmodellen bieden daarvoor een goed hulpmiddel, maar schieten nog op een aantal punten tekort.

Tabel 18.4: Assessment en SCE naast elkaar geplaatst

Software Process Assessment	**Software Capability Evaluatie**
Wordt gebruikt door een organisatie om haar softwareproces te verbeteren	Wordt door een 'inkopende' organisatie gebruikt als selectie-instrument en als middel om de naleving van het contract of project te bewaken
De resultaten zijn uitsluitend bedoeld voor de eigen organisatie	Het resultaat is er voor de eigen organisatie maar ook voor de 'inkopende' organisatie,
De huidige werkwijze wordt 'beoordeeld'	Kijkt naar de huidige werkwijze maar ook naar het potentieel (waar is men mee bezig, wat zijn toekomstplannen?)
De assessment fungeert als een katalysator voor procesverbeterinitiatieven	Er ligt een verplichting tot continu verbeter
Het verschaft de noodzakelijke input voor verbeterplannen en –acties	Analyseert of de ontwikkelaar het in zich heeft de condities in het 'inkoopcontract' na te leven
Zorgt voor saamhorigheid en teamwork; het assessmentteam bestaat uit leden uit de eigen organisatie	Is een onafhankelijk onderzoek, er zitten geen organisatieleden in het evaluatieteam,
Richt zich niet op individuen. Er wordt een 'oordeel' gegeven van de totale organisatie	Wordt gedaan om een specifiek contract binnen te halen
Beoordeling is 'veilig', niet bedreigend.	Kan als bedreigend worden ervaren.

Artikel 16 bij leereenheid 5

Invloed van de factor mens op softwarekwaliteit

Uit: *Softwarekwaliteit*
Fred. J. Heemstra, Rob. J. Kusters en Jos J.M. Trienekens (2002), pp. 253-271

22 Invloed van de factor mens op softwarekwaliteit

22.1 Inleiding

Het wordt steeds duidelijker dat mensafhankelijke factoren bepalend zijn voor de kwaliteit van het product of de dienst die geleverd moet worden. Er kan nog zoveel geïnvesteerd worden in verbetering van processen, methoden en technieken, de investeringen leveren weinig op als de medewerkers die in deze verbeterde processen en met deze nieuwste methoden, technieken en tools moeten werken, niet in staat zijn adequaat hiermee om te gaan of niet bereid zijn hiermee te werken. Gebrek aan deskundigheid, verwaarlozing van opleiding- en trainingsprogramma's , een groot verloop, ziekteverzuim en het niet kunnen binnenhalen van de juiste mensen zijn allemaal factoren die ervoor kunnen zorgen dat goed bedoelde 'improvements' niet opgepakt kunnen worden. Behalve dit soort 'zakelijke' investeringen in medewerkers dient een organisatie ook 'psychologisch' te investeren in mensen. Behalve dat medewerkers niet in staat zijn (Kunnen in figuur 20.2) de potentiële rendementen van verbeteracties te benutten, kan het ook zijn dat medewerkers hiertoe niet bereid zijn (Willen in figuur 20.2). Verbeteracties leiden veelal tot veranderingen in de manier van werken, soms tot reorganisaties. Medewerkers moeten voorbereid en begeleid worden om deze veranderingen te kunnen maken. Vaak gaan veranderingen in werkwijzen gepaard met noodzakelijke gedragsveranderingen. Een softwareafdeling die de draai wenst te maken van een informele, reactieve, adhoc organisatie naar een ISO-gecertificeerde organisatie waarin wordt gemeten, gedocumenteerd en geëvalueerd, vraagt van haar medewerkers een aanzienlijke gedragsverandering. De culturen van beide typen organisaties verschillen enorm.

Van zaken als motivatie, betrokkenheid, gedrag, houding, opleiding, salariëring, toekomstperspectiefen, veilig voelen, etc. mag worden verwacht dat ze een duidelijke invloed hebben op softwarekwaliteit. Maar net als bij organisatieafhankelijke factoren geldt ook voor de meeste mensafhankelijke factoren dat er sterke indicaties zijn voor een relatie, maar dat onvoldoende bekend is hoe die relatie precies is; met andere woorden welk gedrag, welke stimulus, welk opleidingsprogramma e.d. vereist is om een bepaald niveau van kwaliteitsverbetering te realiseren. Vanuit de (organisatie- en gedrags)psychologie worden in meer algemene zin uitspraken gedaan over de invloed van dit soort factoren op de prestaties van professionals. In sommige gevallen kunnen deze bevindingen min of meer rechtstreeks vertaald worden naar de softwareontwikkelomgeving. We hebben echter binnen de softwarewereld nog te weinig zicht op de wetmatigheden van het werk van softwareontwikkelaars om zekere uitspraken te doen over wat wel en wat niet vanuit de psychologie van toepassing is. Voor het management van een organisatie is het een belangrijke vraag hoe zij met ontwikkelaars moeten omgaan, 'aan welke knoppen zij moeten draaien' om de prestaties van ontwikkelaars en daarmee de kwaliteit van het resultaat van hun werk te optimaliseren. Bij de behandeling van de factor motivatie in de volgende paragraaf zal dat dilemma worden toegelicht.

De aanpak die in dit hoofdstuk wordt gevolgd is min of meer identiek aan die in het vorige hoofdstuk. Aan de hand van een bespreking van enkele mensafhankelijke factoren wordt duidelijk gemaakt dat vooralsnog alleen maar uitspraken gedaan kunnen worden in de trant van 'er bestaat een sterk vermoeden dat ...' of 'door ontwikkelaars meer ..... bestaat er een redelijke kans dat...'. Onderzoeksresultaten die aan de hand van empirisch cijfermateriaal kwaliteitsverbetering door gerichte beïnvloeding van factoren als motivatie, gedrag, opleiding, etc. aantonen, zijn schaars. We zullen in paragraaf 22.2 laten zien dat het bijzonder

aannemelijk is dat de kwaliteit van software beter wordt naarmate ontwikkelaars meer gemotiveerd zijn. Het is vervolgens evenwel niet duidelijk waardoor ontwikkelaars gemotiveerd raken, c.q. 'wat hen drijft'. Onderzoeken op dat vlak spreken elkaar tegen. Hetzelfde geldt voor de factor gedrag. Het is enerzijds meer dan aannemelijk dat een bepaald gedrag in een bepaalde situatie leidt tot betere prestaties, maar het is anderzijds onduidelijk welke gedragsveranderingen door het management bij softwareontwikkelaars teweeggebracht moeten worden zodat ze optimaal presteren. In paragraaf 22.3 wordt op deze problematiek nader ingegaan.

Hoewel het lastig is om het verband tussen softwarekwaliteit en mensafhankelijke factoren aan te geven, zullen we enkele handvatten aanreiken. Zo wordt in paragraaf 22.4 het belang van de factor opleiding onderstreept en wordt stilgestaan bij het Personal Software Process-model (PSP). PSP is een concreet opleidingsprogramma, ontwikkeld door Watts Humphrey. Het model heeft als doel de prestaties van de softwareontwikkelaar te verbeteren of analoog aan het CMM, op een hoger niveau te brengen. Het PSP-model heeft als voordeel dat het bijzonder concreet en praktijkgericht is en op korte termijn gerealiseerd kan worden. Dat er ook nadelen kleven aan het PSP wordt in paragraaf 22.4 eveneens toegelicht.

Een factor die in het verlengde van opleiden ligt en die een langere aanloop vraagt, is professionalisering. Opleiden is een van de vele facetten die ervoor zorgt dat de software engineer kan uitgroeien tot een ware professional. In paragraaf 22.5 wordt nader ingegaan op dit onderwerp.

22.2 Motivatie als kwaliteitsbepalende factor

22.2.1 Beschrijving van motivatie

Uit tal van gedragswetenschappelijke studies blijkt dat motivatie van medewerkers en hun betrokkenheid bij het werk tot de belangrijkste factoren behoren die de kwaliteit van het werk c.q. het product bepalen. De hamvraag is echter waardoor professionals gemotiveerd raken, wat hen drijft. Uit het onderzoek van Hackman (1980) komt naar voren dat voor professionals de volgende motivatieprikkels belangrijk zijn:

- *Verscheidenheid aan verlangde deskundigheid*
  Dat wil zeggen dat een professional meer gemotiveerd raakt naarmate er een groter beroep wordt gedaan op zijn deskundigheid en naarmate de activiteiten die hij uitvoert een grotere bijdrage leveren aan de ontplooiing van zijn eigen mogelijkheden.
- *Identiteit van de taak*
  Een professional raakt meer gemotiveerd naarmate hij in staat is een klus van het begin tot het eind uit te voeren c.q. hij verantwoordelijk is voor een complete taak.
- *Belangrijkheid van de taak*
  Voor een professional is het belangrijk dat zijn werk zinvol is en dat zijn werk een wenselijke en merkbare invloed heeft op het leven van andere mensen, zowel binnen als buiten de organisatie. Hoe meer dat het geval is, hoe gemotiveerder hij zal zijn.
- *Autonomie*
  Een professional hecht grote waarde aan vrijheid, onafhankelijk en beslissings-bevoegdheid in zijn werk en vindt het belangrijk dat hij staat wordt gesteld zelf te bepalen hoe het werk wordt ingedeeld, uitgevoerd en over de resultaten wordt gerapporteerd. Hij meer autonomie hij hierin krijgt des te hoger zijn motivatie zal liggen.

- *Terugkoppeling van de taak.*
  Voor een professional is het belangrijk dat hij snel, liefst direct informatie terug ontvangt over de effectiviteit van zijn werk. Naarmate dit beter gebeurt, zal zijn motivatie toenemen.

Door Powell en Posner (1984) wordt benadrukt dat betrokkenheid bij het bedrijf de belangrijkste trigger is om motivatie bij professionals te bereiken. Dit betekent volgens hen dat de organisatie i.c. het management ervoor dient te zorgen dat medewerkers zich committeren aan hun werk, aan de producten, aan de doelstellingen, aan de organisaties. Pas dan kan hoge motivatie een feit worden. Voor het bereiken van een hoge betrokkenheid dient volgens Powell en Posner gestuurd te worden op de volgende variabelen:

- *Visie*
  In het bekende werk van Waterman (*In search of Excellence, 1982*) wordt aangegeven dat excellente ondernemingen zich onderscheiden door een uitgesproken visie hoe het bedrijf te runnen. Deze visie is een gemeenschappelijke goed van alle medewerkers, zij geloven hierin en dragen deze visie uit. De slogan van IBM 'IBM means service' drukt de prioriteit uit die IBM geeft aan de dienstverlening aan haar klanten.
- *Talenten*
  Het onderkennen en benutten van de talenten van medewerkers is een tweede factor die de betrokkenheid verhoogt. Bekend is de formule van de destijds succesvolle softwareorganisatie BSO. Het werk werd georganiseerd in kleine units, medewerkers binnen die units hadden een grote eigen verantwoordelijkheid, hadden volop de mogelijkheid eigen ideeën en creativiteit toe te passen, konden belangrijke beslissingen nemen, etc.
- *Terugkoppeling*
  Terugkoppeling door het management op het geleverde werk is een derde belangrijke factor om de betrokkenheid te verhogen. Net als bij motivatie gaat het niet om een financiële beloning maar om positieve aandacht, om waardering en het geven van het gevoel dat men presteert en een bijdrage levert aan het succes van de organisatie.

Het beeld dat naar voren komt is duidelijk: zelfstandigheid, zinvolheid, verantwoordelijkheid en geïnformeerd worden, zijn voor een professional belangrijke zaken. Wat dat betreft is er niets nieuws onder de zon en bouwt Hackman met zijn onderzoek voort op het bekende motivatieonderzoek van Fred Herzberg (Herzberg, 1959) uit de zestiger jaren. Herzberg gaat uit van het bestaan van zogenoemde 'satisfiers' en 'dissatisfiers'. De eerste groep heeft een positief effect op iemands motivatie, terwijl de tweede categorie dat nou juist niet heeft. In kolom 1 van tabel 22.1 worden deze 'satisfiers en dissatisfiers' weergegeven. Bijvoorbeeld factoren als 'resultaat van het werk' en 'erkenning als professional' zijn de meest dominante factoren, dat wil zeggen zijn het meest motivatieverhogend ('satisfiers') als ze positief worden ingevuld en het meest motivatieverlagend ('dissatisfiers') als er geen of onvoldoende aandacht aan wordt besteed. In een interessant motivatieonderzoek, weliswaar oud maar nog steeds actueel in dit kader, laat Fitz-enz (Fitz-enz, 1978) zien dat de bevindingen van Herzberg en Hackman voor een belangrijk deel eveneens gelden binnen de software engineering. Doel van zijn onderzoek was na te gaan 'waar een softwareontwikkelaar door gemotiveerd wordt, waardoor en waarvoor hij enthousiast wordt' en waarop door het management gestuurd moet worden om de kwaliteit van zijn werk te optimaliseren. Fitz-enz heeft hiervoor onderzocht hoe Herzberg's satisfiers en dissatisfiers van toepassing zijn op softwareontwikkelaars. In tabel 22.1 worden de resultaten van Herzberg en Fitz-enz vergeleken. De doelgroep van Herzberg waren medewerkers uit alle mogelijke hoeken van de

industrie, terwijl het bij Fitz-enz uitsluitend gaat om mensen werkzaam in de softwareontwikkeling.

Tabel 22.1: Vergelijking tussen resultaten van Herzberg en Fitz-enz

Herzberg (industrie)	**Fitz-enz (softwareontwikkeling)**
1. Resultaat van het werk	1. Resultaat van het werk
2. Erkenning als professional	2. Carrièreperspectief
3. Het werk zelf	3. Het werk zelf
4. Verantwoordelijkheid	4. Erkenning als professional
5. Ontplooiingsmogelijkheden	5. Ontplooiingsmogelijkheden
6. Salaris	6. Technisch overzicht, inzicht
7. Carrièreperspectief	7. Verantwoordelijkheid
8. Persoonlijke verhoudingen (in relatie tot ondergeschikten)	8. Persoonlijke verhoudingen (in relatie tot gelijken)
9. Status	9. Persoonlijke verhoudingen (in relatie tot ondergeschikten)
10. Persoonlijke verhoudingen (in relatie tot leidinggevenden)	10. Salaris
11. Persoonlijke verhoudingen (in relatie tot gelijken)	11. Persoonlijke leven
12. Technisch overzicht, inzicht	12. Persoonlijke verhoudingen (in relatie tot leidinggevenden)
13. Beleid van de organisatie	13. Zekerheid van werk
14. Werkomstandigheden	14. Status
15. Persoonlijke leven	15. Beleid van de organisatie
16. Zekerheid van werk	16. Werkomstandigheden

Hoewel er verschillen zijn, zijn deze niet erg in het oog springend. Wat in beide onderzoeken ondermeer opvalt is dat een professional zijn motivatie haalt uit zaken als het werk zelf, het product waarmee hij bezig is en zijn erkenning als professional. Bij softwareontwikkelaars staat het onderwerp carrièreperspectieven duidelijker hoger op de persoonlijke agenda dan bij andere professionals, terwijl salaris weer een beduidend lagere 'satisfier' is.

Splitsen we de resultaten van Fitz-enz op naar functies binnen softwareontwikkeling dan blijken bij sommige motivatiefactoren behoorlijke verschillen te bestaan. Verantwoordelijkheid bijvoorbeeld komt bij programmeurs pas op de negende plaats, bij projectleiders op de vierde en bij managers op de eerste. Ook blijken er duidelijke verschillen te zijn tussen verschillende leeftijdscategorieën en sekse. Wat betreft het verschil in sekse is het interessant te verwijzen naar het eerder genoemde onderzoek van Tracy Hall (1995). Zij komt in haar onderzoek tot de algemene conclusie dat vrouwelijke ontwikkelaars kwaliteit anders beleefden dan mannelijke ontwikkelaars. Deze verschillen kwamen het duidelijkst naar voren als het ging om:

- *de kwaliteit van het werk*
  Vrouwelijke softwareontwikkelaars waren consequent minder tevreden over de kwaliteit van hun eigen werk dan hun mannelijke collegae.

- *de effectiviteit van formele kwaliteitsmethoden en technieken*
  Vrouwen zijn minder overtuigd van het nut en effect van standaards, inspecties en softwaremetrieken.
- *de terugkoppeling over kwaliteit*
  Vrouwen waren meer tevreden over de terugkoppeling vanuit de organisatie over de kwaliteit van het geleverde werk naar het team waarvan ze deel uitmaakten en waren ook duidelijker meer tevreden over de persoonlijke terugkoppeling.

Wat verder opvalt in het onderzoek van Fitz-enz, nog sterker dan bij Herzberg, is dat de motivatiefactoren sterk egocentrisch gericht zijn. Verantwoordelijkheid, nummer 4 in Herzberg's top 5, is de enige factor die gericht is op het belang van de organisatie. Dat egocentrische komt nog sterker naar voren bij de beantwoording op de vraag 'in welke zaken betreffende de organisatie is men (in volgorde van belangrijkheid) het meest geïnteresseerd?'. Tabel 22.2 laat de antwoorden op deze vraag zien. Ook nu weer blijkt dat persoonlijke zaken voorop staan en dan pas worden opgevolgd door zaken betreffende het wel en wee van de organisatie.

Tabel 22.2: Antwoord op de vraag: 'in welke zaken betreffende de organisatie is men (in volgorde van belangrijkheid) het meest geïnteresseerd?'

Persoonlijke interesse van de softwareontwikkelaar
1. Informatie over de huidige prestatie in eigen werk
2. Informatie over toekomstige carrière mogelijkheden
3. Informatie over veranderingen binnen eigen organisatie-eenheid of eigen taken
4. Informatie over het personeelsbeleid dat voor hemzelf is uitgezet
5. Informatie over de winst(gevendheid) van de organisatie
6. Informatie over de organisatie strategie en de uitvoering daarvan
7. Algemene informatie over activiteiten van de organisatie.

Met dit soort gegevens in het achterhoofd kan het management bij de ontwikkeling van software gericht sturen op de motivatie van medewerkers om zodoende de kwaliteit van de software te maximaliseren. Couger (1988) is heel duidelijk in zijn advies richting management:

> *"There is good news for the managers: the number one motivating factor for software developers is the work itself. Software developers don't need a cheerleader – you can concentrate on improving their jobs".*

Hij trekt deze conclusie op basis van een motivatieonderzoek onder 1800 softwareanalisten en programmeurs.

Met dit gegeven in het achterhoofd is het verontrustend om waar te nemen dat uit een zogenoemd 'werknemertevredenheidonderzoek' blijkt dat IT'ers niet erg tevreden zijn met hun werk, in ieder geval minder dan werknemers in de algemene dienstverlening en andere sectoren in het bedrijfsleven (Sijstra, 2000). In tabel 22.3 worden enkele cijfers gegeven. Volgens Sijstra heeft de relatieve lage tevredenheid te maken met het hoge opleidingsniveau van de gemiddelde IT'er en de hoge marktwaarde van zijn specialisme. Uit het onderzoek komt verder naar voren dat de belangrijkste 'satisfier/dissatisfier' de manier van leidinggeven

is waarmee een IT'er wordt geconfronteerd. Sijstra merkt echter op dat elk individu een andere stijl van leidinggeven wenst.

Tabel 22.3: Resultaten van een 'werknemertevredenheidonderzoek'

	Binnen de IT	Buiten de IT
Oordeel werkomstandigheden	6,4	7,3
Oordeel over de werkzaamheden	6,7	7,3
Oordeel over collega's	8,1	8,4
Oordeel over ontwikkelingsmogelijkheden	6,7	5,4

Een soortgelijk onderzoek in Groot-Brittannië geeft ook te denken. Bijna driekwart van de Britse automatiseerders die meer dan 230.000 euro per jaar verdienen, is op zoek naar een nieuwe baan. Van hen staat 63% ingeschreven bij negen of meer verschillende wervings- en selectiebureaus (IT Research House, 2000). Slechts 9% staat bij geen enkel bureau ingeschreven. Een verassing is dat naast zaken als 'een interessante baan' en 'een uitdaging', 'geld' als de belangrijkste motivator worden beschouwd door de Britse IT'ers. Zaken als 'Status', 'Zekerheid' en 'Corporate Benefit' worden nauwelijks op prijs gesteld. Dit wijkt behoorlijk af van de eerder genoemde onderzoeksresultaten.

Als algemeen geaccepteerd is dat een tevreden werknemer een gemotiveerde werknemer is en dat de mate van gemotiveerdheid een belangrijke factor is die de kwaliteit van software beïnvloedt dan geven deze recente onderzoeksresultaten te denken.

22.2.2 Conclusies van motivatie als kwaliteitsbepalende factor

Zoals we in de inleiding al aangaven is de samenhang tussen softwarekwaliteit en motivatie een weerbarstig onderwerp. Uit legio onderzoeken komt naar voren dat er zeker een relatie is tussen beide, maar hoe deze relatie precies ligt, is niet altijd even duidelijk. Uit het onderzoek van IT Research House zou geconcludeerd kunnen worden dat motivatie iets te maken heeft met het persoonlijk gedrag van iemand. Door schaarste op de arbeidsmarkt is een IT'er de afgelopen jaren een schaars goed geworden en heeft de marktwerking ervoor gezorgd met zaken als geld, arbeidsvoorwaarden en carrièreperspectieven een sterke druk wordt uitgeoefend op de IT'er. Het zijn vervolgens sterke schouders die de weelde kunnen dragen en vele IT'ers laten zich blijkbaar verleiden door dit soort satisfiers. Het wel of niet hiervoor openstaan wordt voor een deel bepaald door een diepere schil dan motivatieprikkels en heeft iets van doen met overtuiging, gedrag, levenshouding en dergelijke. Hoewel interessant en zinvol, willen we niet zover gaan om dit soort filosofische onderwerpen verder uit te diepen. Waar we in de volgende paragraaf wel enkele woorden aan willen wijden is de invloed van de factor gedrag op de kwaliteit van software.

22.3 Gedrag en gedragsverandering als kwaliteitsbepalende factor

22.3.1 Beschrijving van gedragskenmerken

Zoals we in deel III hebben kunnen zien is een organisatie in staat haar proces van softwareontwikkeling op een hoger niveau van volwassenheid te brengen door te investeren in

de zogenoemde Key Process Areas. De kans om een hoger niveau te bereiken is groter als niet alleen het proces kwalitatief hoogwaardiger is maar als ook 'de factor mens hoogwaardiger' is (Weinberg, 1994) (Balla, 2000). De uitdrukking 'de factor mens hoogwaardiger' is natuurlijk een eigenaardige en vraagt om enige toelichting. We zullen deze toelichting geven aan de hand van het werk van Berkman (1999) en Weinberg (1994).

Berkman onderscheidt twee vormen van gedrag: gedrag dat hoort bij een organisatie op een lager 'maturity' level en gedrag dat hoort bij een hoger 'maturity' level.
Het gedrag van mensen op een lager 'maturity' level kenmerkt zich als volgt:
- op perceptie gebaseerde acties
- reactief gedrag (firefighting)
- focus op problemen
- weinig gebruik van Key Performance Indicatoren (KPI's)
- weinig verantwoordelijkheden op alle levels
- weinig communicatie.

Dit gedrag heeft tot gevolg dat er geen afstemming tussen de afdelingen is en dat men altijd achter de feiten aan loopt. De gevolgen hiervan zijn grote efficiëntieverliezen en matige of zelfs slechte kwaliteit van het eindresultaat (het product).

Het nieuwe, gewenste gedrag (op een hoger 'maturity' level) is als volgt te omschrijven:
- de te ondernemen acties zijn op data gebaseerd
- de organisatie is pro-actief
- de focus wordt niet meer op problemen maar op oplossingen gelegd
- het gebruik van KPI's wordt gestimuleerd
- verantwoordelijkheid is op lagere niveaus gelegd
- er wordt meer gecommuniceerd en afgestemd tussen afdelingen.

Een gedragsverandering van een lager naar een hoger 'maturity' level is echter niet eenvoudig te realiseren (Berkman, 1999). Investeringen in systemen en productiemiddelen kunnen betrekkelijk snel en eenvoudig zijn, zeker in vergelijking met een gedragsverandering. Een gedragsverandering kun je niet zomaar kopen; het is een complex 'goed'. Als een organisatie wenst te investeren in gedragsverandering als middel om de prestaties te verbeteren, dan moet er eerst een schets worden gemaakt van de huidige gedragssituatie. Er is namelijk een gat tussen de huidige en de gewenste situatie (behavorial gap). Om de 'behavorial gap' te dichten moet, zo vervolgt Berkman, aandacht besteed worden aan de volgende zaken.
- *Visie*
  Iedereen in de organisatie moet werken aan het bereiken van hetzelfde en bekende doel.
- *Vaardigheden*
  Om de doelen te kunnen bereiken moet het personeel beschikken over specifieke vaardigheden. Deze vaardigheden moeten dus bekend zijn.
- *Incentives*
  Zonder incentives zullen mensen hun gedrag niet veranderen. De vraag die mensen bij verandering vaak stellen, is: "wat zit er in voor mij'. In de vorige paragraaf hebben we laten zien dat zo'n incentive niet alleen betekent meer salaris, maar dat ook andere 'satisfiers', zoals interessant werk, ontplooiingsmogelijkheden e.d. belangrijk zijn.
- *Bronnen*
  Onder bronnen kan men tijd, geld, informatie, mensen en faciliteiten verstaan. Vooral tijd is onmisbaar. Immers een gedragsverandering kost tijd, veel tijd.

Met name de wereld van de softwareontwikkeling is er één die voortdurend in verandering is. Proceskwaliteitsmodellen als het CMM, Bootstrap en SPICE propageren zelfs voortdurend veranderen door in een groeimodel door middel van 'continuous improvements' steeds andere en betere werkwijzen toe te passen. Een level 1 organisatie verschilt hemelsbreed van een level 3 of 5 organisatie. Hoewel in het CMM nadrukkelijk gewezen wordt op het belang van gedragsveranderingen bij het beklimmen van de CMM-ladder, wordt in dit soort modellen doorgaans te gemakkelijk van medewerkers verwacht dat zij die groei moeiteloos kunnen meemaken. Te weinig wordt beseft dat hiervoor aanzienlijke gedragsveranderingen nodig zijn.

Weinberg (Weinberg, 1994) onderscheidt niet twee maar zes menselijke gedragspatronen die min of meer corresponderen met het 'maturity' level van de organisatie waarin ze werken. In tabel 22.4 worden deze gedragspatronen met behulp van een paar trefwoorden omschreven en wordt aangegeven welke gedragsveranderingen moeten plaatsvinden om op een hoger level te komen.

Tabel 22.4: Weinberg's gedragskenmerken in relatie tot 'maturity' levels.

Level	**Gedragskenmerk**	**Overgangseis**
1. 'Oblivious'	'we weten zelfs niet dat we bezig zijn met de uitvoering van een proces'	*Nederigheid*; te realiseren door stil te staan bij alles wat je doet'
2. 'Variable'	'we doen wat ons op dat moment het beste lijkt'	*Vaardigheid*; te realiseren door training en ervaring
3. 'Routine'	'we werken volgens vaste regels (behalve wanneer we in paniek raken)'	*Stabiliteit*; te realiseren door Quality Software Management
4. 'Steering'	'we kiezen uit alle regels voorschriften op basis van het resultaat dat we ermee kunnen bereiken'	*Waakzaamheid*; te realiseren door tools en technieken
5. 'Anticipating'	'we hebben regels en voorschriften gebaseerd op onze ervaring ermee'	*Aanpasbaarheid*; te realiseren door persoonlijke ontwikkeling
6. 'Congruent'	'iedereen is betrokken bij het doorlopend verbeteren van alles'	

Weinsberg's onderzoek maakt duidelijk dat gedrag en met name het vermogen om het gedrag aan te passen een belangrijke factor is om iemands prestatie en de kwaliteit van zijn werk te verhogen.

22.3.2 Conclusies van gedrag als kwaliteitsbepalende factor

De onderzoeken van Berkman en Weinberg maken duidelijk dat niet alleen een organisatie moet groeien naar een hoger 'maturity' level om betere software te kunnen maken, maar dat ook een persoon parallel aan dit groeiproces moet meegroeien naar een hoger 'maturity' level. En net zoals een organisatie een geleidelijk, stap-voor-stap groeitraject doorloopt, zo zal ook het gedrag van een persoon een soortgelijke, geleidelijke en stap voor stap groeitraject moeten doormaken. Er is nog maar weinig bekend hoe het gedrag van software engineers in de 'goede' richting veranderd kan worden. We hebben wel een idee van een gewenst gedrag maar tasten nog vaak in het duister hoe dit gewenste gedrag te realiseren. Daar komt bij dat door veranderende visies op het ontwikkelwerk zelf de ideeën over gewenst gedrag niet stabiel zijn en dat gedragsveranderingen veel, heel veel tijd vragen. Net zoals bij de factor motivatie geldt voor de factor gedrag dat er meer dan aannemelijk een verband is tussen softwarekwaliteit en gedrag, maar dat de stand van de wetenschap vooralsnog tekort schiet om hier zekere uitspraken over te doen.

Zeker is dat gedragsveranderingen alleen mogelijk zijn als de voorwaarden om te kunnen veranderen aanwezig zijn. Een van die voorwaarden is de beschikbaarheid van voldoende kennis en kunde om op een hoger niveau te kunnen opereren. In de volgende paragraaf zullen we stilstaan bij de relatie tussen opleiden en de kwaliteit van software. Voordat het zover is, willen we aan de hand van een kort intermezzo het belang van kennis en kunde, kortom van een goed opgeleide ontwikkelaar illustreren.

22.4 Het belang van kennis en kunde

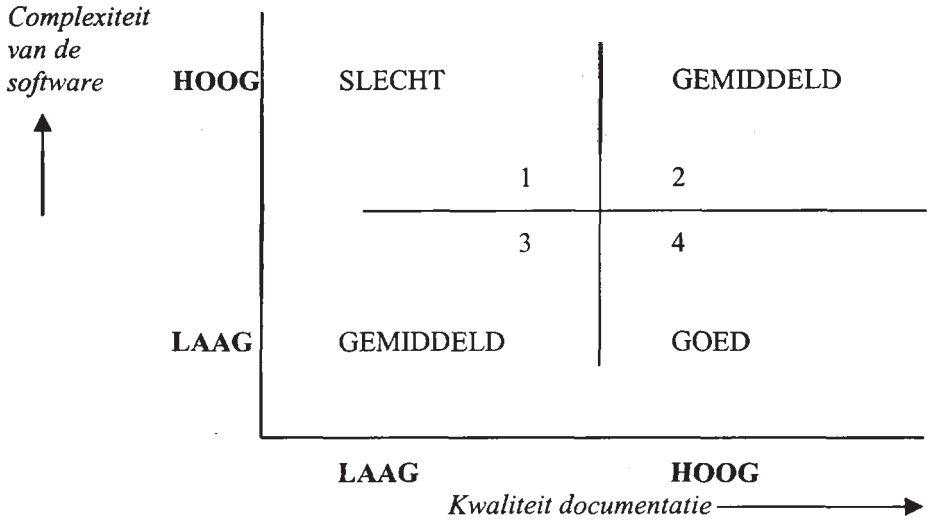
Het effect van opleiden, wordt ondermeer geïllustreerd door Les Hatton (1998). Hij geeft aan dat de vakkennis van een softwareontwikkelaar een bepalende factor is om snel fouten in de software te vinden, te analyseren en te verbeteren. De snelheid van vinden, analyseren en verbeteren is volgens Hatton afhankelijk van onder andere:

- de complexiteit van de software en
- de kwaliteit van de documentatie.

Een van de dilemma's van de software engineering is dat veel ontwikkelaars ervan uitgaan dat de door hen ontwikkelde software goed gedocumenteerd en goed ontworpen is, terwijl bij het optreden van fouten het tegenovergestelde blijkt. De software is veel complexer dan men denkt, is vaak beperkt gedocumenteerd en slecht ontworpen. Goede, dat wil zeggen volledige en eenduidige documentatie dient garant te staan voor bijvoorbeeld begrijpelijke foutmeldingen. Op basis van een goed ontwerp wordt complexe software inzichtelijk en 'leesbaar'. Hierdoor is het mogelijk de oorzaak van een fout snel te vinden. Door complexiteit van de software en kwaliteit van de documentatie aan elkaar te relateren ontstaan vier combinaties zoals weergegeven in figuur 22.3.

De meest beroerde situatie treedt op in het kwadrant 1 'complexiteit hoog, kwaliteit laag'. We hebben dan te maken met onbegrijpelijk foutmeldingen en bovendien met software die nauwelijks toegankelijk is. Hatton noemt het volgende vermakelijke voorbeeld:

> ***"error -23009: there are already more than 64 tcp or udp streams open {tcp:104}"***
> *This appeared on the screen of a new Macintosh g3 desktop running Apple's latest and greatest version of OS 8. Two desolate hours later the person realised that an acceptable substitute might have been: "modem not responding".*



Figuur 22.3: De relatie tussen kwaliteit van documentatie en complexiteit van software

De ideale situatie is er een waarbij de foutmelding rechtstreeks wijst naar de bron van de fout en de software zo goed is gestructureerd en geordend dat moeiteloos de fout gealloceerd kan worden. Een voorbeeld van een situatie, kwadrant 4, met 'complexiteit laag, kwaliteit hoog' is (Hatton 1998):

> ***"dereference pointer contents 0x0 at***
> ***Strlen(...) Called from***
> ***Line 126 of myc_constexpr.c called from***
> ***Line 247 of myc_evalexpr.c called from***
> ***Line 2459 of myc_expr.c"***
> *It points unerringly at the responsible code line and usually takes a matter of moments to fix.*

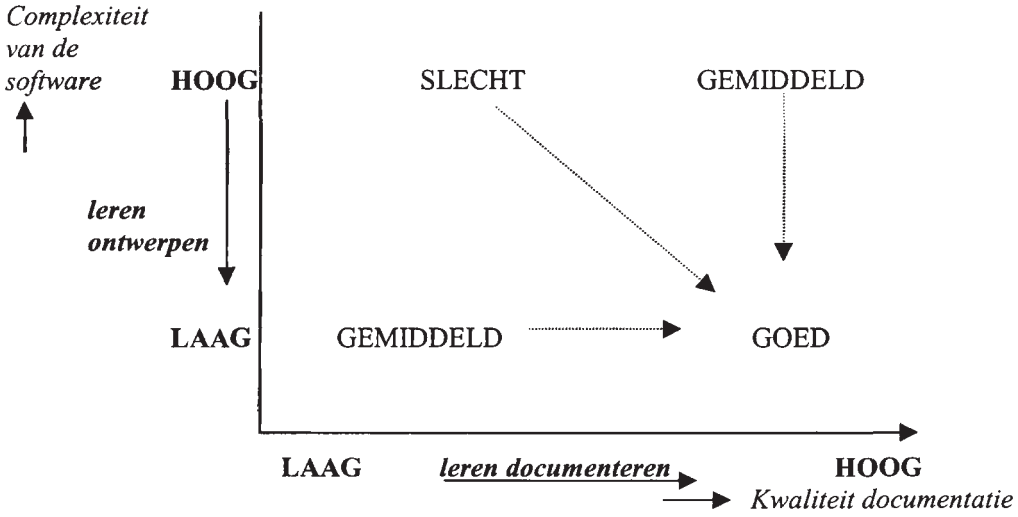
Hatton geeft talloze voorbeelden van de gevolgen van matige documentatie in combinatie met matig ontworpen software (kwadrant 3). Een bijzonder vermakelijk voorbeeld is het volgende:

> ***"system stressed ..."***
> *This appeared on the cash registers of the author's local pub. An hour of exciting discussion about communication protocols, deadlocks and such later, one realised an acceptable substitute might have been: "printer out of paper".*

Om dit soort problemen te voorkomen pleit Hatton voor maatregelen om:
- de documentatie van software te verbeteren
- de complexiteit van software te verminderen.

Een van de belangrijkste maatregelen om het eerste doel te bereiken is het beter opleiden van ontwikkelaars zodat zij in staat zijn documentatie te schrijven die anderen kunnen begrijpen. Foutmeldingen dienen ondubbelzinnig te zijn en precies aan te geven om welke fout het gaat en welke actie nodig is om de fout te herstellen. Om de complexiteit te verminderen pleit Hatton voor maatregelen die leiden tot een beter ontwerp. Door ongestructureerd ontwerpen is de kans bijzonder groot dat in de kluwen van code niet meer is na te gaan waar wat staat en waar een foutmelding naar verwijst. Ook hiervoor geldt dat door middel van opleiding de

ontwikkelaar geleerd moet worden gestructureerd te ontwerpen. Het effect van beide opleidingsmaatregelen wordt in figuur 22.4 weergegeven.



Figuur 22.4: Op weg naar eenvoudige en goed gedocumenteerde software

De illustraties van Les Hatton benadrukken het belang van opleiding, kennis en kunde. In de volgende paragraaf willen we aangeven op welke wijze opleiden ingevuld kan worden. Hiervoor kunnen legio mogelijkheden worden aangedragen. We zullen ons beperken tot het opleidingsprogramma dat door het Personal Software Process-model wordt voorgeschreven. De keuze voor dit model is gemaakt omdat PSP een duidelijke relatie heeft met het uitvoerig behandelde CMM en het nog te behandelen Team Software Process (TSP) model. Daarbij komt dat door Humphrey, de ontwikkelaar van PSP, gemeten is naar het effect van de toepassing van PSP op softwarekwaliteit. Kortom, er is empirisch evaluatiemateriaal beschikbaar.

22.5 Personal Software Process (PSP)

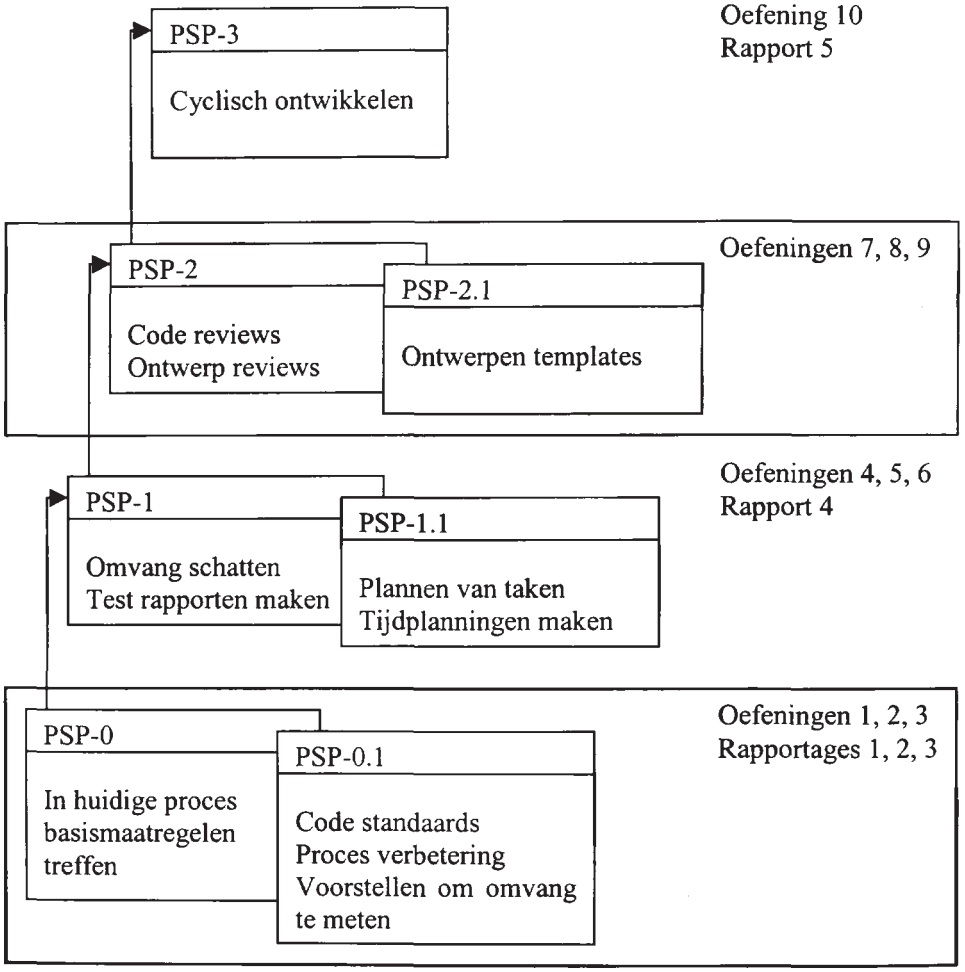
Watts Humphrey (1996) geeft een korte en duidelijke omschrijving van zijn PSP-model:

> *"The Personal Software Process' is a framework of techniques to help engineers improve their performance – and that of their organizations – through a step-by-step, disciplined approach to measuring and analyzing their work".*

Volgens Humphrey (2000) zijn de vruchten van het gebruik van PSP bijzonder zoet: minder fouten in de code, betere schattingen en planningen en hogere productiviteit. Bovendien is PSP vanaf CMM level 3 een noodzakelijk vehikel om procesverbeteringen mogelijk te maken. Een andere reden voor Humphrey om PSP te ontwikkelen was de constatering dat het CMM vooral binnen grote organisaties weerklank vond en dat het CMM voor kleine softwareorganisaties iets was als 'schieten met een kanon op een mug'.

Zoals uit figuur 22.5 blijkt biedt PSP een vastomlijnd leertraject dat een softwareontwikkelaar moet doorlopen. Door in totaal tien leeropdrachten uit te voeren en vijf rapporten te schrijven doorloopt de ontwikkelaar de verschillende PSP levels (van PSP-0 naar PSP-3). Stap voor

stap wordt zijn inzicht vergroot en krijgt hij handvatten om betere software op een betere wijze te maken.. Uiteindelijk moet een ontwikkelaar dan in staat zijn zijn eigen werk op een gedisciplineerd wijze uit te voeren. Een centraal thema binnen PSP is het meten van de eigen prestatie. Na iedere stap c.q. leeropdracht dient de ontwikkelaar het effect van het geleerde te meten en te evalueren. In figuur 22.5 wordt de structuur van PSP weergegeven.



Figuur 22.5: De structuur van PSP

Zonder al te gedetailleerd in te gaan op het PSP-leertraject, kunnen de verschillende PSP levels met daarin opgesloten de PSP-concepten als volgt globaal worden omschreven:

Personal Management
In deze eerste stap leert de ontwikkelaar hoe hij bepaalde PSP-formulieren en -werkwijzen in zijn eigen werk moet toepassen. Hij doet dit door ontwikkeltijd en aantal fouten te meten. De ontwikkelaar moet dus gegevens verzamelen. Met deze gegevens kan hij nagaan of hij in de loop van de tijd vorderingen maakt (zijn kwaliteit beter wordt). De stap Personal Management bestaat uit drie fasen: planning, ontwikkeling (ontwerp, codeer, compileer en test) en postmortem. In de eerste fase krijgt de ontwikkelaar a) een standaard die hem voorschrijft hoe te coderen, b) een metriek om de omvang te meten en c) een zogenoemd 'Process

Improvement Proposal' formulier. Met behulp van dit formulier houdt de ontwikkelaar bij welke problemen optreden en legt hij ideeën vast hoe hij in het vervolg zijn eigen proces kan verbeteren. De ontwikkelaar leert bovendien wat het nut is van het verzamelen en vastleggen van gegevens.

Personal Planning
In deze stap wordt een methode geïntroduceerd waarmee de ontwikkelaar op basis van zijn eigen verzamelde gegevens de omvang en de ontwikkeltijd van zijn programma's kan inschatten. Verder krijgt hij geleerd hoe hij plannen moet opstellen en hoe plannen worden opgedeeld in taken en activiteiten. Door vroeg in het ontwikkeltraject te plannen, leert de ontwikkelaar het belang om al snel gegevens te verzamelen. Op deze wijze ervaart hij het nut van statistische schattings- en planningsmethoden.

Personal Quality
In deze stap komt het omgaan met fouten aan de orde. Aan de hand van de opgetreden fouten in zijn eigen programma's leert de ontwikkelaar checklists te maken en te gebruiken om ontwerpen en code te reviewen. De ontwikkelaar leert waarom het belangrijk is meteen vanaf het begin te letten op kwaliteit en leert hoe hij op efficiënte wijze reviews kan uitvoeren. Hij ervaart ook dat hij de checklists sneller kan aanpassen naarmate zijn vaardigheden en ervaringen toenemen. In deze stap worden ontwerpspecificatie en analysetechnieken geïntroduceerd, samen met fout preventie technieken, procesanalyse en benchmarktechnieken. Door te meten hoe lang hij bezig is met een taak en het aantal fouten dat hij veroorzaakt en oplost, is de ontwikkelaar in staat om zijn eigen prestaties te evalueren en te verbeteren

Scaling Up
Dit is de laatste stap in de PSP-aanpak. De ontwikkelaar wordt nu geconfronteerd met steeds lastiger en omvangrijke opdrachten. Hij krijgt nu geleerd wat onder andere ontwerp verificatie methoden zijn.

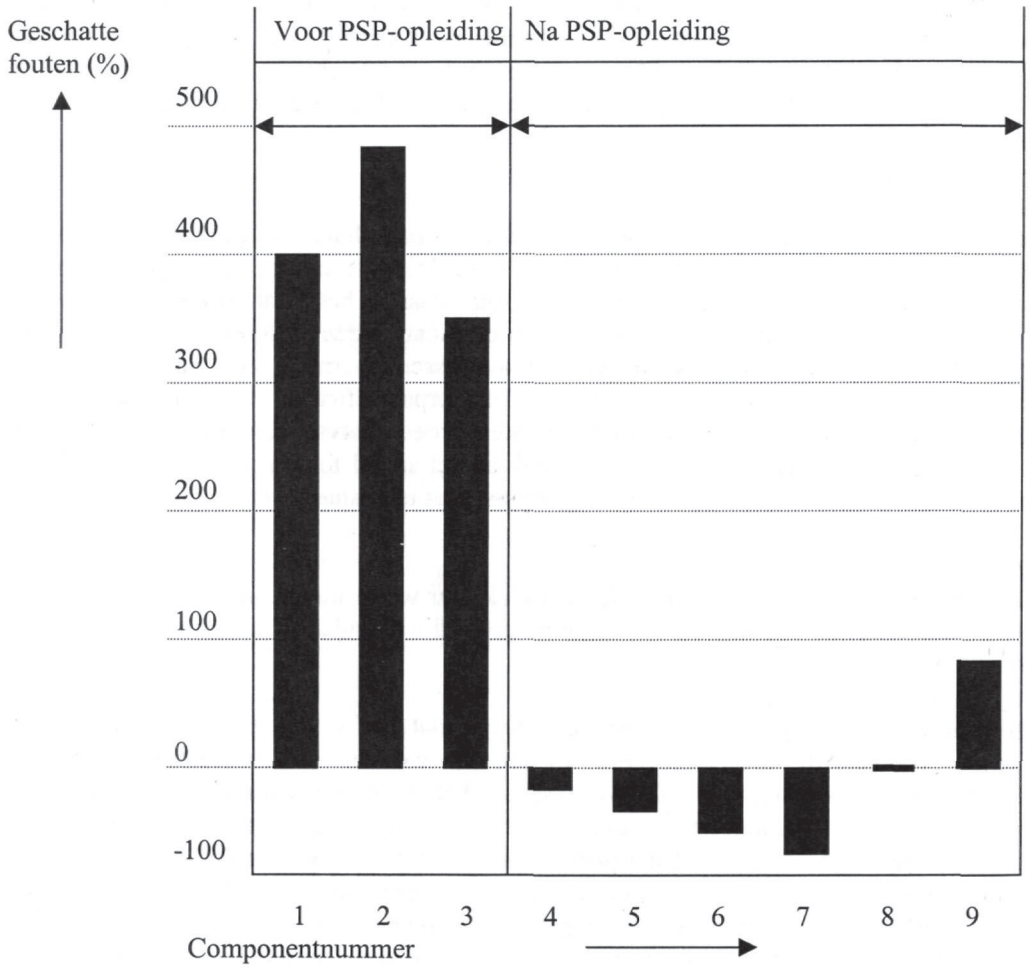
We hebben in het vorige deel kunnen constateren dat het CMM handvatten biedt c.q. activiteiten voorschrijft die uitgevoerd dienen te worden om procesverbetering op organisatieniveau te realiseren. Het CMM verschaft dus mogelijkheden om het werk goed uit te voeren. Het CMM garandeert dat evenwel niet. Om de mogelijkheden van het CMM optimaal te kunnen benutten, moet het proces d.w.z. de activiteiten die het CMM benoemt, uitgevoerd worden door een ontwikkelaar die specifieke kennis, vaardigheden en een gedisciplineerde werkwijze in huis heeft. Op dit punt komt PSP in beeld. PSP gaat over het hanteren van de principes van procesverbetering op het individuele niveau van de ontwikkelaar met als doel sneller en goedkoper, betere software te maken. Om alles uit PSP te halen, moet de ontwikkelaar werken in een omgeving die hem die mogelijkheden biedt. Vandaar dat PSP het hoogste rendement heeft in softwareorganisaties die zich in de buurt van of boven het CMM level twee bevinden.

Het CMM en PSP bevruchten elkaar wederzijds. Het CMM biedt die gestructureerde en geordende omgeving die ontwikkelaars nodig hebben om hun werk zo goed mogelijk te doen. PSP levert ontwikkelaars die in staat zijn topkwaliteit te leveren mits zij kunnen werken in een daarvoor aangemeten omgeving.

PSP wordt nog niet, zoals het CMM, op grote schaal toegepast. We zien echter dat het aantal organisaties dat overgaat tot toepassing van PSP geleidelijk toeneemt (bijvoorbeeld Baan, Boeing, Motorola en Terdyne (Humphrey, 1998). Evaluatiegegevens van gebruikers laten

duidelijk de voordelen van het PSP-leerprogramma zien (Humphrey, 1996). Naast beter schatten en plannen en een hogere productiviteit wordt door PSP geschoolde ontwikkelaars betere software geleverd.

In figuur 22.6 worden in gecomprimeerde vorm de evaluatiegegevens van een softwareontwikkelteam bij het bedrijf Advanced Information Services gegeven (Ferguson e.a., 1997).



Figuur 22.6: Evaluatiegegevens van toepassing van PSP

Halverwege het ontwikkeltraject kreeg het team een PSP-opleiding. In de figuur worden de relatieve schattingsfouten van voor en na de opleiding in beeld gebracht. Voor component 1 bijvoorbeeld, werd de ontwikkeltijd oorspronkelijke geschat op 4 weken. In werkelijkheid bleek de benodigde tijd 20 weken te zijn. Een onderschatting van dus 394 %. Na de PSP-opleiding bleken de ontwikkelaars veel realistischer schattingen te kunnen afgeven. In tabel 22.5 worden de testtijden van een ontwikkelteam vergeleken voor en na het ondergaan van een PSP-leertraject (Furgeson e.a., 1997).

Tabel 22.5: Besparingen bij het testen door gebruikmaking van PSP

Testtijden voordat de PSP training was gegeven		
Project	**Omvang**	**Testtijden**
C	19 requirements	3 testcycli
D	30 requirements	2 maanden
H	30 requirements	2 maanden
Testtijden voordat de PSP training was gegeven		
Project	**Omvang**	**Testtijden**
B	24 requirements	5 dagen
E	2.300 regels code	2 dagen
F	1.400 regels code	4 dagen
G	6.200 regels code	4 dagen
I	13.300 regels code	2 dagen

De eerste resultaten van het gebruik van PSP zijn hoopgevend (zie ook Kamatar en Hayes, 2000) en Zong e.a. (2000). Enkele kritische opmerkingen zijn echter op zijn plaats. Allereerst zijn de evaluatiegegevens gepubliceerd door de ontwikkelaars van PSP. Het wachten is op de resultaten van andere, wellicht objectievere evaluatoren. De eerste resultaten uit een andere dan de SEI-omgeving laten zien dat de gemaakte investeringen de baten ver overtreffen. Bovendien leidde het gebruik van PSP niet tot minder fouten die door de gebruiker werden gemeld (Morisio, 2000) .Een tweede opmerking is principiëler van aard. Hoewel er positieve elementen te vinden zijn in PSP, gaat het enigszins voorbij aan het gegeven dat ontwikkelwerk voor een deel een creatief en intellectueel proces is. Het rigide, enigszins mechanistische mensbeeld van een ontwikkelaar dat schuil gaat achter de filosofie van PSP is hiermee strijdig.

Het voordeel van het PSP ten opzichte van bijvoorbeeld acties die moeten leiden tot gedragsveranderingen of hogere motivatie bij softwareontwikkelaars is dat het opleidingsprogramma van PSP uiterst concreet en praktisch is. In plaats van een lange aanlooptijd zoals bij gedragsveranderingen het geval is, kan met het PSP morgen aan de slag gegaan worden en zijn de effecten op korte termijn zichtbaar en meetbaar. Bovendien is opleiden c.q het PSP een opstap naar een professionalisering van het vakgebied software engineering en haar beoefenaars. Maar voor professionalisering is meer nodig dan alleen kennis, kunde en opleiding. Een zekere mate van volwassenheid, een begrip dat al herhaaldelijk in dit boek is genoemd, is een voorwaarde. Voor een jonge discipline als de software engineering is dat een lastig obstakel. In de volgende paragraaf geven we enkele bespiegelingen over dit onderwerp.

22.6 Professionalisering

Dietz (1999) heeft met zijn L_PASO-model een grote stap voorwaarts gezet in het realiseren van professionaliteit van het vakgebied software engineering. Dietz begint met de constatering dat op de weg naar professionaliteit meteen al een lastige hobbel ligt, namelijk de onoverzichtelijkheid van het werkterrein, de 'jungle' aan beroepen en functies. Wie weet wat een system support analist is of een informatie-architect of een network-consultant, laat staan

dat men een notie heeft wat de taken zijn die bij deze functies horen. Het is zondermeer duidelijk dat de jonge discipline software engineering geen heldere, bruikbare en stabiele structuur heeft. Dat maakt de herkenning en ook de erkenning van de software engineer, vergeleken met andere beroepsgebieden, extra lastig. De eerste stap op weg naar professionaliteit in elk vakgebied en dus ook dat van de software engineering is volgens Dietz het ondubbelzinnig en nauwkeurig definiëren van de begrippen die voor het bereiken en bewaken ervan relevant zijn. Het L_PASO-model biedt een structuur om die relevante begrippen in de software engineering te definiëren. Op basis van die begrippen die het werkterrein van de software engineer in kaart brengen, worden competenties beschreven die een software engineer dient te bezitten en de prestaties die met die competenties geleverd moeten kunnen worden. L_PASO geeft verder aan hoe de beroepsbeoefenaar zijn competenties kan verhogen en in het verlengde daarvan zijn professionaliteit kan verbeteren. Aangegeven wordt hoe, welke competenties verworven kunnen worden en hoe deze gecertificeerd kunnen worden.

Het L_PASO-model start met het aangeven van de soorten systemen waarmee de software engineer in zijn werk te maken heeft. Het betreft de volgende drie systemen, waarvan de eerste twee uitvoering ter sprake zijn gekomen in dit boek:
1. bedrijfsystemen
2. informatiesystemen
3. infrastructurele systemen.

Bij het bestuderen van een systeem kan de software engineer twee, tegengestelde, oriëntaties innemen. Deze zijn:
1. *functie-oriëntatie* (gericht op de functionaliteit/extern gedrag van het systeem)
2. *constructie-oriëntatie* (gericht op de constructie/interne werking van het systeem).

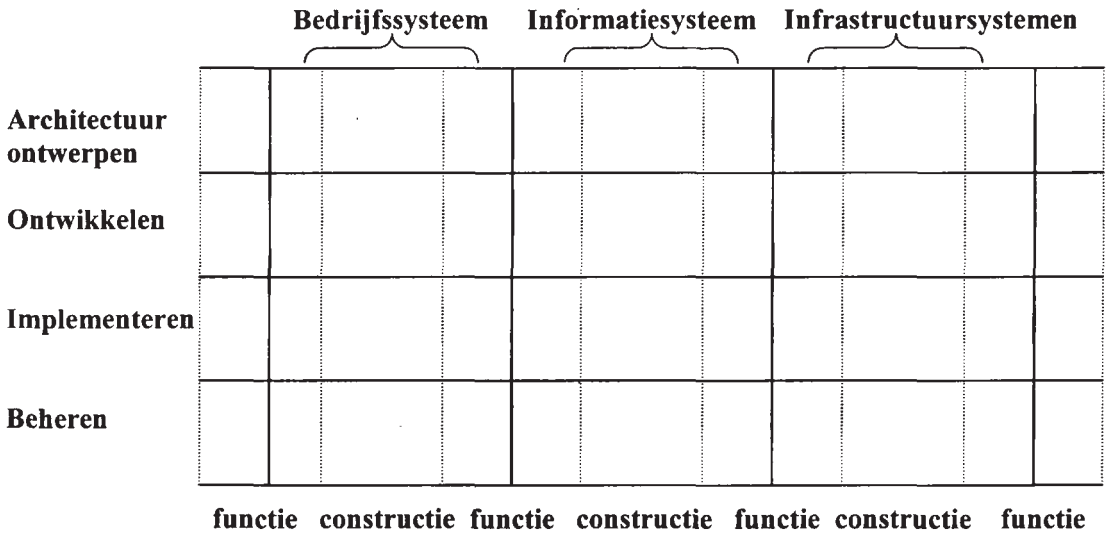
De activiteiten die door de software engineer worden verricht met betrekking tot de hiervoor genoemde soorten systemen, kunnen worden ingedeeld in de volgende vier groepen:
1. *architectuur ontwerpen*
2. *ontwikkelen*
3. *implementeren*
4. *beheren.*

Door deze drie begrippen (systemen, orientaties en activiteiten) onderling te relateren ontstaat het domein van de software engineering, waarbij elke combinatie van een activiteitsoort, een systeemsoort en een systeemoriëntatie een elementair stukje is van het domein (zie figuur 22.7)

In zijn boek gaat Dietz uitvoerig in op alle domeinen. We volstaan hier met het noemen van de overall structuur en een verwijzing naar het betreffende boek.

Als het gaat om prestaties en competenties maakt het model wat betreft prestaties een onderscheid in de volgende rollen:
- Uitvoering
- Management
- Advisering
- Auditing
- Onderwijs.

Door elk domein te combineren met deze rollen ontstaan zogenoemde werkelementen. Al het werk dat een software engineer doet bestaat uit een aantal van deze werkelementen. Bij competenties wordt een onderscheid gemaakt in competentiesoorten en competentieniveaus, die onderling gecombineerd competentie-elementen opleveren. Het is nu zaak om werkelementen en competentie-elementen goed op elkaar af te stemmen. Het L_PASO-model geeft aan hoe een en ander gebeurt.



Figuur 22.7: Het domein van de software engineer

Binnen de IEEE Computer society and the Association for Computing Machinery wordt eveneens gewerkt aan de ontwikkeling van een professioneel vakgebied (Bourque, 1999). Zo is er een werkgroep actief met het in kaart brengen van de zogenoemde 'Software Engineering Body of Knowledge' (SWEBOK) (http://www.swebok.org). Zonder zo'n gemeenschappelijke visie en omschrijving van Software engineering en wat wel en wat niet tot het vakgebied behoort, is volgens de werkgroep professionalisering niet mogelijk, kunnen er geen eenduidige diploma's worden afgegeven en kunnen er ook geen accreditaties voor opleidingen worden toegekend. In de Body of Knowledge wordt een onderscheid gemaakt tussen de volgende acht belangrijke aandachtsgebieden, waarvan softwarekwaliteit overigens er een is.
Aandachtsgebieden:
- softwareconfiguratiemanagement
- softwareconstructie
- softwareontwerp
- software engineering-infrastructuren
- software engineering-management
- software engineering-proces
- software-evolutie en -onderhoud
- softwarekwaliteit
- software requirements-analyse
- softwaretesten.

Zoals al eerder opgemerkt, is het L_PASO-model een goed stap in de richting van professionalisering van de software engineering. Temeer ook omdat het model op verzoek van

de beroepsvereniging van informatici is ontwikkeld. Het model is een eerste stap, gebruiken en verder uitbouwen van het model is een tweede stap. Voor zo'n tweede stap is de aanwezigheid van een krachtige beroepsvereniging een belangrijke voorwaarde. We zullen enkele woorden wijden aan beroepsverenigingen en aangeven wat hun bijdragen is aan de professionalisering van de software engineering. We zullen ons beperken tot de volgende verenigingen:

- de Vereniging van Registerinformatici
- 'Council of European Professional Informatics Societies'
- 'Professional Development Scheme'.

De VRI (Vereniging van Registerinformatici) heeft als haar missie geformuleerd 'het bevorderen van het kwaliteitsniveau waarop in Nederland de beroepsuitoefening door Registerinformatici plaatsvindt'. De hiervan afgeleide doelen zijn:

- het verhogen van het kwaliteitsniveau waarop in Nederland de beroepsuitoefening door registerinformatici plaatsvindt,
- het verenigen van informatici op grond van opleidings- en ervaringseisen en de gedragscode.

Voor het bereiken van deze doelen ontplooit de vereniging een aantal activiteiten, waarvan we enkele belangrijke willen noemen:

- toelating van informatici op grond van door de VRI gestelde criteria
- toetsing van naleving van de gedragscode en afhandeling van klachten over de naleving ervan
- initiëren van onderzoek en ontwikkelingen op het gebied van de kwaliteit binnen het vakgebied.

De gedragscode is een belangrijk instrument dat door de VRI wordt voorgeschreven. De code luidt als volgt:

> *'Bij mijn handelen als informaticus zal ik steeds het belang van de samenleving in al haar facetten positief dienen. Ik heb mij daarom laten registreren in het Register van Informatici. Ik geef daarmee te kennen dat ik als zodanig publiekelijk herkenbaar wil zijn en aangesproken wil worden op de gedragscode.'*

Deze gedragscode wordt vervolgens vertaald in een groot aantal gedragsregels, die hier niet verder ter sprake zullen komen. Hiervoor verwijzen we naar de VRI.

De 'Council of European Professional Informatics Societies' (CEPIS) heeft als missie het bevorderen van de professionalisering van het vakgebied informatie & communicatietechnologie (ICT). Om deze missie te realiseren heeft de CEPIS zich als doelen gesteld om

1. de basis vaardigheden en kennis te definiëren waaraan een IT professional (in Europa) dient te voldoen en
2. te komen tot een internationaal erkend certificaat.

De twee fundamenten van de CEPIS zijn verwoord in:

1. de 'European Informatics Skills Structure', waarin van de werkgebieden binnen de ICT standaards worden gegeven van de benodigde kennis en vaardigheden
2. het 'European Informatics Continuous Learning Programme', dat gebruikt kan worden om via een self-assessment te onderzoeken welk leertraject uitgezet dient te worden.

Zonder verder in details te treden, willen we wijzen op de Amerikaanse tegenhanger van de CEPIS, namelijk de American Society for Quality (ASQ). Voor uitgebreidere informatie verwijzen we naar Hamilton (1998).

De 'Professional Development Scheme' (PDS) is een professionaliseringsprogramma dat door de British Computer Society (BCS) in het leven is geroepen om kwaliteitsbeheersing te bevorderen en via opleiding en scholing carrièremogelijkheden binnen de software engineering te bieden.
Enkele basiselementen van het PDS zijn:
- een 'Industry Structure model' waarin een uitgebreide beschrijving te vinden is van kwaliteitseisen die worden gesteld aan personen die werkzaam zijn binnen de software engineering
- een jaarlijkse inspectie door de BCS om na te gaan of de voorschriften worden nageleefd.

22.7 Conclusies

In dit hoofdstuk hebben we, net als bij behandeling van organisatieafhankelijke factoren, laten zien dat onder de noemer 'mensafhankelijk' talloze factoren te benoemen zijn waarvan aangenomen mag worden dat ze een belangrijke invloed hebben op de kwaliteit van software. Voor veel factoren geldt dat het verband aannemelijk is, maar voor weinige kan dat ook onderbouwd worden aangetoond. Dat neemt niet weg dat investeren in die factoren waarvoor zo'n relatie met softwarekwaliteit aannemelijk is, zinvol is. Vanuit de gedragspsychologie wordt bijvoorbeeld voldoende bewijs aangeleverd om ervan uit te aan dat motivatie en gedrag zondermeer zeer belangrijke beïnvloedende factoren zijn. De vraag 'hoe gedrag te beïnvloeden en hoe motivatie te verhogen?' blijft lastig te beantwoorden. Een eensluidend antwoord kan vooralsnog niet worden gegeven. Vanuit de psychologie is bekend dat zaken als:
- *verscheidenheid aan verlangde deskundigheid*
- *identiteit van de taak*
- *belangrijkheid van de taak*
- *autonomie*
- *terugkoppeling van de taak*

belangrijk zijn voor de motivatie van een professional en in het verlengde daarvan van een softwareontwikkelaar. Hoewel dit soort onderzoeksresultaten vaak worden herkend, blijkt de praktijk van alledag weerbarstiger te zijn en blijken ook zaken als salariëring en carrièrekansen belangrijke motivatieprikkels te zijn.

Anders ligt het bij zaken als opleiden en professionalisering. Hiervoor kunnen concrete acties worden afgesproken en het positieve effect ervan op softwarekwaliteit is duidelijker aantoonbaar. De eerste resultaten met toepassing van het opleidingsprogramma volgens het Personal Software Process-model van W. Humphrey zijn hoopgevend. Initiatieven zoals het L_PASO-model van Dietz en de IEEE (SWEBOK) om te komen tot een 'common Body of Knowledge' moeten het vakgebied software engineering helpen de weg naar volwassenheid te vinden.

Artikel 17 bij leereenheid 5

The Darker Side of Metrics
Douglas Hoffman

Software Quality Methods, LLC, 2000

The Darker Side of Metrics

Douglas Hoffman, BACS, MBA, MSEE, ASQ-CSQE
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
doug.hoffman@acm.org

Abstract

There sometimes is a decidedly dark side to software metrics that many of us have observed, but few have openly discussed. It is clear to me that we often get what we ask for with software metrics and we sometimes get side effects from the metrics that overshadow any value we might derive from the metrics information. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when metrics are applied. I believe most people in this field are hard working and well intentioned, and even though some of the behaviors caused by metrics may seem strange, odd, or even silly, they are serious responses created in organizations because of the use of metrics. Some of these actions seriously hamper productivity and can effectively reduce quality.

This paper focuses on a metric that I've seen used in many organizations (readiness for release) and some of the disruptive results in those organizations. I've focused on three different metrics that have been used and a few examples of the behaviors elicited in organizations using the metrics. For obvious reasons, the examples have been altered to protect the innocent (or guilty).

Biography

Douglas Hoffman is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for over 25 years and now is a management consultant specializing in strategic and tactical planning for software quality. He is Section Chairman for the Santa Clara Valley Section of the American Society for Quality (ASQ) and is past Chairman of the Silicon Valley Software Quality Association (SSQA). He is also a member of the ACM and IEEE, and is certificated by ASQ in Software Quality Engineering and has been a registered ISO 9000 Lead Auditor. He has earned an MBA as well as an MS in Electrical Engineering and BA in Computer Science. He has been a speaker at dozens of software quality conferences including PNSQC and has been Program Chairman for several international conferences on software quality.

The Darker Side of Metrics[1,2]

Introduction

Software measures and metrics have been around and used since the earliest days of programming. I have studied and used software measures and metrics with varying degrees of success throughout my career. I might even be labeled a reformed measurement enthusiast[3]. During the 25 years or so that I have studied and used software metrics I have been surprised by some of the effects the metrics have had on the organizations, and often I have been extremely distressed over the negative impacts I have seen. Even though I have touted software metrics and successfully begun several metrics programs, every software organization I have observed that has used metrics for more than a few years has had bizarre behaviors as a result. There is a decidedly "dark side" to these metrics programs that impacts organizations all out of proportion to what is intended. In the last year Kaner[4,5,6] has provided a framework for understanding why this might occur. One source comes from a lack of relationship between the metrics and what we want to measure (Kaner's 9th factor)[7] and a second problem is the over-powering side effects from the measurement programs (Kaner's 10th factor)[8]. The relationship problem stems from the fact that the measures we are taking are based on models and assumptions about system and organizational behavior that are naïve at best, and more often just wrong[9]. Gerald Weinberg provides excellent examples in his *Last Word* article analyzing some benign software inspection metrics[10]. Weinberg shows how counting defects found during preparation and at code inspections gives metrics relating mostly to the number of inspectors and telling almost nothing about the product or process it proports to measure.

It is clear to me that we often get what we ask for with software metrics. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when

---

[1] This information was first generated for presentation and discussion at the *Eighth Los Altos Workshop on Software Testing* in December, 1999. I thank the LAWST attendees*, Chris Agruss, James Bach, Jaya Carl, Rocky Grober, Payson Hall, Elisabeth Hendrickson, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Quoc Nguyen, Bret Pettichord, Melora Svoboda,* and *Scott Vernon,* for their participation and ideas.

[2] I differentiate between the measures of an attribute and metrics computed from the measures. Ultimately we should take measures to compute metrics.

[3] Lawrence, Brian "Measuring Up," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[4] Kaner, C. "Rethinking Software Metrics," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[5] Kaner, C. "Yes, But What Are We Measuring?," 1999 PNSQC

[6] Kaner, C. "Measurement of the Extent of Testing," 2000 PNSQC

[7] ibid.

[8] ibid.

[9] Many models go so far as to ignore mathematical truths. Many times we categorize based on ordinal scales; Defect Severity, for example. We assign numbers to the categories and depict the order based on the values we chose. We know that a "Severity 1" isn't ½ as much as a "Severity 2," and we can't claim that all "Severity 3" defects are the same. We could just as well use colors and call the categories as Green, Yellow, Orange, and Red. Doing arithmetic with them (e.g., the Priority is Severity times Likelihood) is as absurd as multiplying colors.

[10] Weinberg, G. "How Good Is Your Process Measurement," *Software Testing and Quality Engineering* vol. 2, no. 1 (2000)

metrics are applied. Don't take me wrong; I believe most people in this field are hard working and well intentioned. Although some of the behaviors cause by metrics may seem funny or even silly, there are potentially serious consequences to organizations because they use metrics. The specific observations I make here are based on real companies using software metrics in their product development. I have taken some care to change enough of the details that the innocent (or guilty) cannot be easily identified. In some instances, I have combined observations from multiple organizations. But, you wouldn't be alone if you think you recognize your organization in some of the situations. I've noticed that people often recognize their own experiences here.

Three Metric Examples

I've selected three examples of metrics used to decide when a product is ready to release. There certainly are other examples and other metrics, but this has been a particularly ripe area of examples from my experience. The three metrics used to show a product's readiness for release are:

1. Defect find/fix rate
2. Percent of tests running/Percent of tests passing
3. Complex model based metrics (e.g., COCOMO)

Briefly, each of the metrics is used to describe an attribute of project status (how far along is the project, is it ready for release, are we meeting our milestones, etc.). These attributes were applied by management to monitor and adjust project plans and member behaviors in order to keep the project on schedule. I haven't a clue about the attributes' scales and don't think anyone else can, either. The variation in the attribute and the measures is all over the map – a few projects run like clockwork (or so I've heard), but most don't run as planned, and some I've worked with were just out of control. (Out of control is a term I use for

Kaner's Ten Measurement Factors

1. The purpose of the measure. What the measurement will be used for.
2. The *scope of the measurement*. How broadly the measurement will be used.
3. The *attribute to be measured*. E.g., a product's readiness for release.
4. The appropriate *scale for the attribute.* Whether the attribute's mathematical properties are rational, interval, ordinal, nominal, or absolute.
5. The *natural variation of the attribute.* A model or equation describing the natural variation of the attribute. E.g., a model dealing with why a tester may find more defects on one day than on another.
6. The *instrument that measures the attribute.* E.g., a count of new defect reports.
7. The *scale of the instrument*. Whether the mathematical properties of measures taken with the instruments are rational, interval, ordinal, nominal, or absolute.
8. The *variation of measurements* made with this instrument. A model or equation describing the natural variation or amount of error in the instrument's measurements.
9. The *relationship between the attribute and the instrument*. A model or equation relating the attribute to the instrument.
10. The *probable side effects* of using this instrument to measure this attribute. E.g., changes in tester behaviors because they know the measurement is being made.

software that has progressively much worse quality as developers try to patch it up, followed by project cancellation or quick turnover of most of the management and staff.)

I've neaver heard of any direct measure of project status, program readiness for release, or progress toward meeting milestones, etc. Instead, we've used surrogate measures and metrics; the instruments we used to measure are:

1) counters of new and resolved defect reports,
2) percents of tests running and passing, and
3) a "Mulligan's stew" of metrics (including cyclomatic complexity, defect counts, defect find/fix rates, defect severities, estimated size of programs, experience levels of developers, past projects' metrics, and others) combined and mixed thoroughly in an arithmetic equation (such as COCOMO).

Defect find/fix rate

The first two metrics use pairs of measures to determine convergence on the planned project completion. The ratio of defects found to defects fixed intuitively feels like a reasonable way to see the end. When we find more than we fix (ratio greater than 1) during a specified time period, we are discovering problems faster than fixing them. When the ratio equals 1, we are not gaining ground or losing it in terms of fixing problems. When the ratio gets below 1, the developers are reducing the number of known problems. For the life of the project, the ratio of all defects found to all defects fixed should approach 1 as we fix all known problems. This model is based on several assumptions that don't hold:

1) all defects that are found are reported,
2) there is a goal of fixing (or resolving) all known defects,
3) when all known defects are fixed the product is ready to release,
4) there are reasonable resolutions for all fixed defects.

Defect counts don't naturally vary, given a consistent definition of defects. One has either been found and reported or it hasn't. The count of the number of reported defects can easily be done, and several people are likely to come up with the same counts given the same defect database. However, human nature makes hash of the numbers by accident and with purpose. Reducing the effort to hunt new defects or withholding reports will directly and immediately improve the ratios. Developers and testers can become extremely creative in recategorizing defects as enhancement requests, problems in other products, duplicates, unassigned, etc. in order to resolve them without fixing the underlying problems.

A few examples of observed behaviors of these sorts may serve to clarify:

- To reduce the number of defects, twenty-five reports against a subsystem were all marked as "duplicates" of one new defect. The new defect report referred to each of the twenty-five for a

description of the problem (because the only thing the twenty-five had in common was that they were reported against the same subsystem).

- In an organization where defects didn't get counted before initial screening and assignment, a dozen defects that hadn't been resolved in more than four weeks were assigned to the developer "Unassigned," and thus were not counted.
- In one case the testers withheld defect reports to befriend developers who were under pressure to get the open defect count down. In another case the testers would record defects when the developers were ready with a fix to reduce the apparent time required to fix problems.
- A test group took heat for not having found the problems sooner (to give the developers more time to fix the problems).
- Developers only reported problems after they had been fixed (thus never making the ratio worse).
- I've seen defects fall in the crack, get lost, pushed in circles, or be forever deferred.

One general reaction to found/fixed metrics was the creation of "Pocket lists" of defects by developers and testers. Developers kept these unofficial lists of defects and action items to themselves. If they reported the defects, it created a negative perception about the code and they also needed to address the problems. One manager went so far as to publicly criticize individuals for having more than five defects against their modules. The project (with 40 modules) now *never* has more than 200 defects reported (and seldom fewer than that). The pocket list has been as benign as not reporting problems observed and fixed in code, and as blatant as knowing about existing problems others were likely to encounter.

There are also ways that management can make this situation much worse (such as a Dilbertian "bug bonus" for the number of bugs found or fixed), so the developer and tester are encouraged to report and fix large numbers of defects kept in pocket lists so they create the appearance of a flurry of last moment heroic activity. Indeed, it is quite difficult to take software measures without creating significant side effects.

Percent of tests running/Percent of tests passing

The second case (percent of tests running and percent passing) also may intuitively feel like a reasonable way to see the end. If 100% of our tests run, and 100% pass, we're done, right? The percentage of tests running can be interpreted two ways; either the testers haven't had time to run all the tests, or the software isn't complete enough to fully test. Likewise, the percent of tests that pass may feel like a reasonable way to measure progress. The terms themselves are difficult to pin down, and no matter how they are defined and enforced, there are simple ways to manipulate the percentages. Also, this model of testing makes several false assumptions:

1) all tests are known before testing begins,
2) whatever constitutes "a test" is well understood and agreed upon,
3) whatever constitutes "running a test" is well understood and agreed upon,
4) test outcomes are clearly pass or fail, and

5) release occurs when all tests (or a specified percentage) pass.

Very few organizations I've worked with do only pre-defined tests. Most test groups mix regression tests with exploration and continue to create new tests until the product escapes. Unless we know the exact number of tests we will run, we don't know the denominator. And, the definitions of a test, running a test, and passing and failing are subject to debate and manipulation. If a test is used in several configurations, is it a separate "test" in each? If it's only counted as a test once, do failures in several (but not all) configurations count as one failure, several, or a fraction of one? Does a test fail twice if it detects more than one defect? Does a long exercise count as multiple tests? Do we only count those tests that are for completed features? Do we have to run tests we know will fail? Do we have to report defects against those failures?

Some examples of observed behaviors due to these metrics:

1) the redefinition of what a "test" is in order to increase the number to be counted and increase the percentage passing. (Each test is divided into sections because having one of ten (1000 line) tests that can't run looks much worse than three of two hundred (50 line) tests.)
2) top management declares victory and releases because "All four of the tests that could run were tried, and 100% passed. (The code just wasn't complete for the other 5,768 tests.)"
3) replacement of expected results with actual (bad) results because a problem was "known." (Development demanded that testing remove the test from the count of tests running and not passing for those defects that weren't going to be fixed. Management would not let the testers reduce the count of tests running, so they compromised…) [I call this "institutializing a defect" – making sure it stays in the product forever.]

Complex model based metrics

The last situation uses complex multivariate mathematical models to describe the project. A "mathematical" model is applied to decide ahead of time how many defects there should be in modules. The models are also used to predict the rate of defect reporting, so the progress and readiness of a project can be discerned simply by counting the defects found to date. "According to the model, we should find 50% of the defects through our testing. Since there are 200 defects in the project (according to the Function Point computations), the project should be ready to release when we've found 100 defects." These predictions then become self-fulfilling prophecies, with sometimes crippling side effects.

For these measures, there isn't any real relationship between the measured or subjectively assigned attributes and the meaning assigned. There might be a relationship between the mathematical model and organizational behavior or project status, but I doubt it. The amazing thing I've observed is the near religious fervor that goes into defending the validity of the model based on the fact that on the last *N* projects, the equation has yielded a precise estimate of release on the day of release. (Never mind that it wasn't correct any of the 52 weeks previous to that, and the subjective values and equation fudge factors were changed every one of the past 52 times the equation was used.) This is what I call the "you always find your keys at the last place you look for them" effect. When they look back at the project,

they conclude they now know the numbers for "experience factors," "product complexity," and all the other elements that plug into the equation. They only really know that numbers can be chosen for the equation to show what they now know – its ready for release. The sad thing is the number of managers and engineers who don't realize that given any moderately complex equation with multiple variables, values can be selected to generate any particular result. For example, given the equation 5 * X + 3 * Y + Z, we can pick values of X, Y, and Z to yield any positive result.

Some examples of observed behaviors due to these metrics:

- Managers demanding sign off by testers without testing because the model showed release should occur in spite of testing not being complete. ("We've followed the curve precisely for eight months – obviously it's ready for release.")
- Punishment of testers for not finding enough defects quickly enough through lowering their rating (and thus their pay).
- Reporting of minor problems, variations on a defect, and seriously questionable tests, to increase defect counts.
- Testers not reporting defects (or reporting trivial problems) in order to keep the defect counts corresponding with the predicted values from the model. (Because management had such faith in the models, people were expected to perform exactly as predicted. Having too many defects was interpreted as meaning the project was poor quality and too few was interpreted as the tester doing a poor job.) [I've been given the argument that "the exactly predicted number of defects was reported every week on an 18 month project." This is a statistical impossibility for a real world process. W. Edwards Demming described the real world effect as being due to normal random variation. My chemistry instructors called too perfect data in an experiment "dry-labbing," documenting the predicted results instead of taking accurate measurements (even subconsciously).

Other Side Effects

In response to noticing some of the above side effect behaviors, the rules can change. In some situations, defect fixes cannot be deferred to future releases because opening a project with defects already reported skews the statistics (and anyway, the argument goes, the defects weren't put in on this project; they were put in on previous projects). But, since products cannot be released with known defects and the defects cannot be deferred, they are resolved ('not a defect' or 'no fix') in the current project and may be reentered if someone remembers them during the next project.

Deferral was used as a technique to reduce the number of defects, so management mandated justification in person for all deferred defects. Consequently, the number was reduced through consolidation of defects (25 marked as duplicates of a new one that references each of the 25). Then all duplicates required management review. Defects were then resolved en mass as "no fix intended." The rules changed then to force management review of all defect reclassification. Then defects became stacked into "submitted/need assignment" to keep them from getting into the statistics until resolution was ready.

Conclusions

Software metrics have been successfully employed for decades to understand, monitor, and improve products and processes. There are volumes of literature describing successes and methods, and organizations regularly implement metrics programs. In software development and quality assurance there is almost blind acceptance of the value of such programs, even though in many of these same organizations the metrics program is secretly causing lower productivity and quality.

It is imperative for any organization interested in quality to be alert and careful about metrics. Even organizations that have well established programs, especially organizations with long established metrics programs, ought to consider whether the metrics have the desired meanings and identify what side effects are caused. Where efforts are diverted without improving the product or its quality, some questioning should be made as to the appropriateness of the measures and metrics. The unintended side effects may be slowing rather than streamlining the organization, and can even serve to obscure our understanding of test results and reduce the overall product quality.

Cem Kaner has provided a framework for understanding and rethinking software metrics, but observations of behaviors within organizations is often sufficient to recognize unintended side effects. By reassessing the meanings of our metrics and recognizing their limitations we can potentially reduce the negative impacts.