Statistical Process Control for Software Development – Six Sigma for Software revisited

Dr. Thomas Fehlmann

Euro Project Office AG
Zeltweg 50, CH-8032 Zürich, Switzerland
thomas.fehlmann@e-p-o.com
www.e-p-o.com

Abstract. Six Sigma is a popular management strategy for the shop floor. Can it be applied to software? Software development is knowledge acquisition – it is different from industrial engineering. This paper explains the mathematical foundations for statistical thinking when lack of data samples prevents the use of proper statistics. The approach is based on Deming's process chains and Akao's Quality Function Deployment (QFD), using its ability to track requirements consistently through a complex structure of functional topics, and enhanced by Six Sigma metrics for both the quality of goal setting as for execution. The investigation of the mathematical foundations for QFD yields a surprising explanation how QFD and Six Sigma statistics are connected.

1 Introduction

When we write Software, we use processes. We have an input, usually customer requirements or – at least – expectations, we expect output such as code, a computer system running some new applications, or services, and we have a set of controls such as time, cost, and other quality attributes. Thus writing software resembles at least in theory a process, as we know from industry. Nevertheless, statistical process control, which has proved to be so useful in many other industries, seems to fail completely, when applied to software engineering.

1.1 The Nature of Software Development

Software development is knowledge acquisition – it is different from industrial engineering. Industrial engineering is a transition from an imagination into reality – first is the idea, then the detailed specification, the build and testing, improving the construction if needed until it works, then operation.

Software is different. Initially, we have but a rough idea of what we want, then we must think about it in more detail, find out what resources we need, what technology to use, what the constraints are in terms of time and cost. The more we know about the problem, the better we can define it and break into parts, the closer we get to the solution.

1.2 Statistical Process Control

In production, statistical process control is a technique based on collecting data from the output that the process produced, such as physical quality attributes like size, weight, or form, and measure variations. By comparing those measured variations with various other process parameters, root causes for variations become apparent. Eliminating root causes limits these variations. We iterate that until all variations encountered remain within tolerable limits. Such limits we call *tolerance interval*. Process results that lie outside of the tolerance interval we call *defects*. A defect therefore is a strange behavior of the process results that affects the customer or user.

1.3 The Metrics for Variance

From statistics, the Six Sigma approach uses the following metrics for variance:

$$\sigma = \sqrt{\frac{n * \sum x_i^{\,2} - \left(\sum x_i\right)^2}{n^2}} \qquad \textbf{(1)}$$

where n is the size of the data sample, and x_i are the measured data components, for i=1…n. The variance σ corresponds approximately to $^1/_6$ of the bell's curve base length, if the data distribution is statistically independent. This is why we say, if σ is small enough to fit six times into the tolerance interval, our process has Six Sigma.

1.4 Statistical Process Control for Software

Trying to apply this paradigm to software development, we encounter many difficulties. The first problem is, that there is an important time gap between the software development process and the time the software is being used. When testing software, it is difficult to assess which kind of behavior the future users will perceive as strange, and thus rate as a defect. During tests, we only can compare with specifications or requirements, and unwanted behavior detected in reviews or tests are called *bugs*, and usually we try to remove bugs before they can be delivered to customers, who will eventually see them as defects.

However, neither requirements nor specifications are usually detailed enough, we don't know how many defects they contain, and perceptions change over time. Even if we put a lot of effort into strong and detailed specifications and testing during the creation phase of the software life cycle, the bugs detected do not behave statistically "correct". They tend to be everything but statistically independent from each other (for instance, see Fenton [6]).

The next problem is, that it is very difficult to understand what a defect is in software. According Six Sigma, a defect affects the customer's ability to use the software. We distinguish two kinds of failure possibilities:

- *A-defects*, i.e. requirements not detected or nor understood; and
- *B–defects*, i.e. bugs in the delivered functionality.

Some bugs delivered to customers do not affect their ability to use the software, because there is redundancy and a work-around exists. Therefore, it is not even clear whether such failures are B–defects. A–defects, in turn, are even more difficult to detect. The users, without explanation, might reject using software with A–defects.

Writing software requires a highly disciplined approach, just as industrial engineering does. Processes for software development are necessary, but their output is hardly predictable with standard statistical process control.

2 Passing Knowledge from One Function to Another

Knowledge is the ability to relate things together. Understanding the cause for one phenomenon enables mankind to predict the likely outcome of similar phenomena – weather forecast is one example, predicting global warming is another one. Parents and schools pass such knowledge to their children in order to help them better to survive and become more successful.

Writing software is similar: developers pass such knowledge to machines. The only particularity is that developers usually know their machines better than the topics they try to teach them. Thus developing software is not just the work of individuals, it is rather a chain of processes that eventually end in some useful software.

2.1 Deming's Process Chain

For software, we typically distinguish three kinds of process areas [4] that matter:

1. *Decision processes* – including the market, competition and organizational needs. Usually the deciders materialize in the form of customer requirements, explicitly or implicitly stated needs of the customer. We use the term *customer's needs* to refer to the results of the decider's processes
2. *Enabling processes* – process capability and maturity enabling an organization to write software. Those process kinds typically result in a certain *capability* profile.
3. *Realization processes* – software has to go through a number of processes that transform the customer's needs into design, business scenarios and use cases. In turn, each of those part results connects with its specific test: design with integration test, business scenarios with application tests, and use cases with functional tests.

Fig. 3 contains a sample Deming process chain for software development.

Processes are interdependent. The output of one process constitutes the input of the subsequent process. In the context of software development, we use the name *functional topics* for input and output of the development processes. Functional topics depend from the specific software development processes in use; it might be different whether we develop scenario-based applications that support users in their business, or rather software that controls mechanical or logical functions. In addition, decision processes depend from whether software is developed in-house for internal purpose, or rather as a product or service component.

2.2 A Formal Notion of Knowledge – Defining Knowledge Acquisition

For two functional topics G and X, *knowledge* is the set of cause/effect relationships, which characterize how X relates to G. We need this formal definition for speaking about knowledge acquisition.

The functional topics consist of *requirements*. A functional topic may have a potentially unlimited number of requirements. We write $g \in G$ when g is some requirement for the functional topic G.[1]

Requirements are bound to functional topics. User requirements and the many kind of technical requirements are different. The failure to understand that difference ([11], [19]) is the main reason why software development processes are difficult to manage.

The generic term "cause" also needs some explanation what it means for software development. The "cause", why software supports a requirement, is the set of requirements in the function topic one step below. For instance, a certain number of Use Cases supports a required business scenario. Each Use Case, in turn, depends from the classes that implement the functionality required in the Use Case. Knowledge embraces all possible *solution requirements* needed for a goal requirement with regard to the respective functional topics. As an example, take a business transaction for a goal requirement, and the use cases that prepare, execute, and assess the results of that transaction as solution requirements.

Knowledge is always limited and finite, even if functional topics are potentially infinite. The knowledge $X \rightarrow G$ about a process consists of a finite set of *knowledge terms* $\{x_1, \ldots, x_m\} \rightarrow g$ where $x_1, \ldots, x_m \in X$ and $g \in G$; m being a finite number.[2]

The knowledge term $\{x_1, \ldots, x_m\} \rightarrow a$ is the formal representation of an *Ishikawa diagram*, where the requirements $b_1, \ldots, b_m$ of functional topic G are the solution requirements for the requirement a in functional topic X.

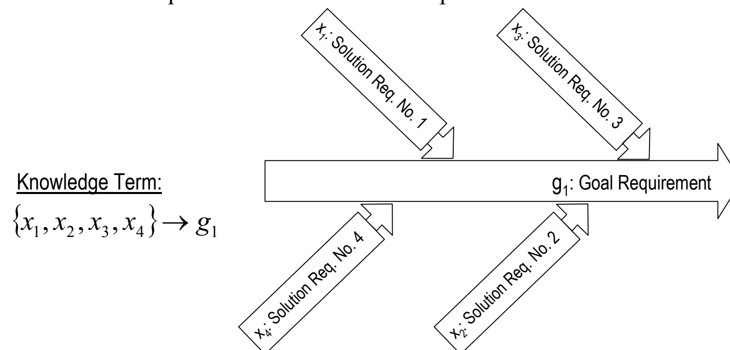

Fig. 1. The Ishikawa diagram, or fishbone diagram, is a tool for cause/effect analysis. It provides a method for visualizing the causes that produces the effect; in our case associating the solution requirements to the goal requirements that they support

---

1 Those readers familiar with set theory may think of a being an element of set G when reading $g \in G$. In this understanding, the functional topic G consists of all its true statements.

2 Knowledge Terms are the famous Arrow Terms of Engeler [4] and have been introduced as a foundation for Quality Function Deployment in [7] and [8]. For software, we prefer the notion of "knowledge" rather that the more formal "Arrow".

The solution requirements $x_1, \ldots, x_m$ are not equally weighted, because they do provide specific contributions to the goal. It is a common practice to distinguish three different levels of cause/effect relationship: weak, medium, and strong. Strong relationship means that the respective solution requirement is a strong influence factor for the respective goal requirement; this corresponds to weight 9. Medium relationship means, it is an indispensable service but might be used in other contexts too; this gives weight 3. "Weak" means useful but not vital, and we assign weight 1. No cause/effect relationship corresponds to weight zero[3].

To the knowledge term $\{x_1, \ldots, x_m\} \rightarrow g$ we assign the scalar weights $\alpha_1, \ldots, \alpha_m$, where the weights α_j range in $\{0, 1, 3, 9\}$, and each scalar α_j describes the strength of relationship between the cause x_i and the effect g.

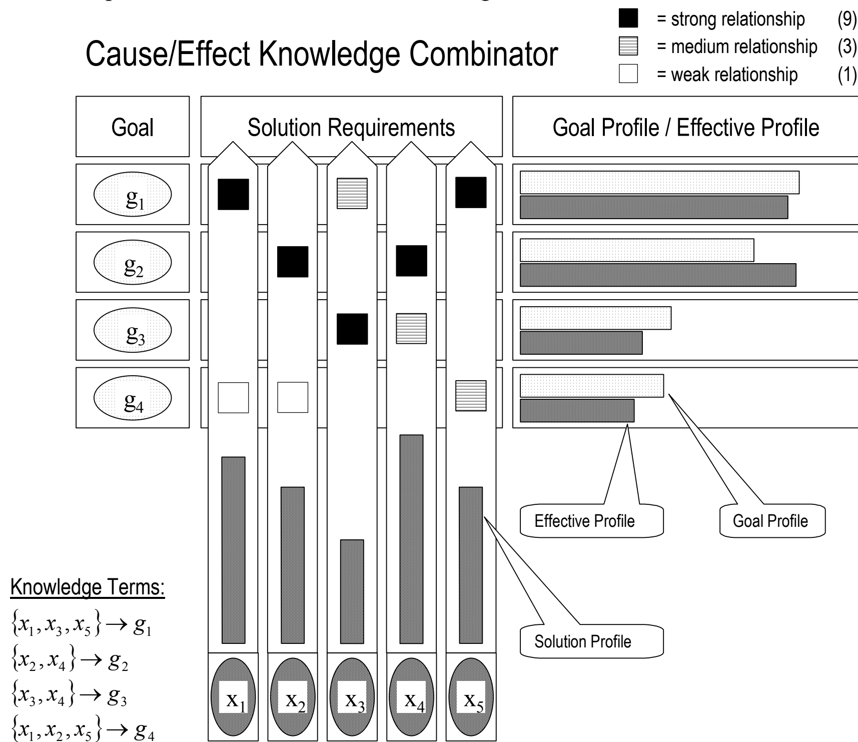

Fig. 2. Generic Cause/Effect combinator relating requirements $\langle g_1, \ldots, g_4 \rangle$ with solution approach $\langle x_1, \ldots, x_4 \rangle$

Now assume that you have a set of n requirements $\{g_1, \ldots, g_n\}$ from functional topic G, and for each representative g_i some knowledge terms $\{x_{i,1}, \ldots, x_{i,m}\} \rightarrow g_i$, assuming $i=1, \ldots, n$.

Let $\alpha_{i,1}, \ldots, \alpha_{i,m}$ be the corresponding relationship weights. We can arrange them in a $n \times m$–matrix:

[3] See [2], [10], [16], and [23] for a discussion of variations of that practice.

$$\varphi = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \dots & \alpha_{i,j} & \dots \\ \alpha_{n,1} & \dots & \alpha_{n,m} \end{bmatrix} \quad (2)$$

where $\alpha_{i,j} \in \{0, 1, 3, 9\}$ for i=1, …, n and j=1, …, m

We call the matrix φ of relationship weights a *knowledge combinator* (see Fig. 2). Six Sigma knows about such matrices as the method of Quality Function Deployment (QFD) [2]. QFD is widely used as the vital part of "Design for Six Sigma", see for instance [17], [18], and [22]. According a communication of Prof. Akao, the QFD matrices were invented originally as a convenient form to combine several Ishikawa diagrams for the same functional topic at once.

Well-established techniques exist for characterizing functional topics with only a few requirements [21], [10]. By choosing comprehensive requirements for functional topic G, you can keep the number of requirements low for that functional topic and thus describe Deming processes by just a few characteristic requirements on both the input and the output side.

2.3 Topic Profiles – Measuring Knowledge

With QFD, we have to possibility to measure knowledge acquisition and its variation along the Deming process chain. This constitutes the basic idea behind "Design for Six Sigma". We do not need to count knowledge in some way. It is sufficient to study the process itself and its results, the requirements.

Let G be a functional topic as before. The topic G may represent the Customer's Needs (CN) that result from the decision processes in the Deming process chain. The requirements of G are not equally graded; there exist different weights. If the effects are not equal, the solution requirements cannot be either. Thus for the solution requirements X in the knowledge X→G there must also exist profiles that give suitable weights to the requirements of X. In practice, this means that the resulting effect on G depends both from the selection of requirements in X, and from their respective weights. Such weight distributions we call *(requirement) profiles*.

Usually you have a goal profile for the requirements $g_1, \dots g_n \in G$; this is the scalar vector $<g> = <\gamma_1, \dots \gamma_n>$. This vector represents the profile of the desired effects.

Given any solution requirements $x_1, \dots x_m \in X$, we would like to know its solution profile that produces an effect as close as possible to the goal requirements (see Fig. 2). Assume the profile for the cause vector[4] is $<x> = <\xi_1, \dots, \xi_m>$, the ξ_j representing scalar values for the weights of the solution requirements x_j, j=1, …, m.

The matrix (2) allows for the calculation of the resulting profile when applying knowledge X→G to the functional topic X. Then we calculate the weights of the resulting effects as follows:

[4] We use the square brackets <…> to distinguish vectors from scalars.

$$\varphi(\langle x \rangle) = \langle \varphi_1, \dots, \varphi_n \rangle = \left\langle \sum_{j=1}^{m} \alpha_{1,j} * \zeta_j, \dots, \sum_{j=1}^{m} \alpha_{n,j} * \zeta_j \right\rangle \quad \textbf{(3)}$$

component wise: $\varphi_i = \sum_{j=1}^{m} \alpha_{i,j}$, i=1, …n

Again, the $\alpha_{i,j}$ denote the QFD matrix elements, which represent the weights of the relationship between solution requirements and effects (i=1, …, n; j=1, …, m); φ is a mapping from a vector with m components into another vector with n components. We call the resulting profile $\varphi(\langle x \rangle) = \langle \varphi_1, \dots, \varphi_n \rangle$ the *effective profile*. Unfortunately, the effective profile will not automatically match the original goal profile <g>. With <x> selected arbitrarily, there is a gap between the vectors <g> and φ(<x>). The only commonality between the two vectors is that they both have the same number of components, namely n.

Formula (3) allows assessing the effects of measured topics, back up the process chain. For instance, we can measure the functional size [1], [12] of the specified software that is the result of the Use Case (UC) process (see Fig. 3). This is knowledge acquisition of the form UC→BS, where we must find the best selection of use cases that meet the requirements for the Business Scenarios (BS). Let <u> be the functional size profile for a representative set of use cases. Using the knowledge combinator β from UC→BS, their effective profile is $\beta(\langle u \rangle) = \langle \beta_1, \dots \beta_n \rangle$, according formula (3).

The effective profile β(<u>) tells us the relative cost distribution for the business scenarios. This technique is very effective; it solves the problem for many CIO how he should value ICT services.

However, this is not all. We can track the effective profile β(<u>) yet another step back and, using the knowledge combinator from BS→CN, find out which customer's needs receive most attention by the functionality provided. Thus, we can track the use cases' cost profile back to the value perceived by the customer. Such transparency avoids spending effort for features and functions that yield no value for the customer. For product managers, this information is of indispensable.

It takes idling out of the software development process. Similarly, we may measure test results, compute defect density and track measurement results back to see whether what we tested was of any importance to the customer.

2.4 Comparing Profiles – Analyzing Knowledge

However, how do we know whether our knowledge combinators are accurate enough to allow for such backtracking? We need a metric that tells us how well our knowledge terms models the software development processes. For this, we need statistical process control for the requirement profiles.

To compare vectors, it is not sufficient to compare their difference. Although you can compute the difference vector as soon as you have the same amount of components, the result may be useless unless the components of the two vectors are of comparable size. In order to achieve that, we need *normalization*.

When drawing a profile, it will not change its graphical appearance when multiplying each coefficient by the same number; only the size of the graphics will adapt. In order to compare two profiles, we need to gauge the profiles to the same size.

The *length* of a profile $\langle g \rangle = \langle \gamma_1, \ldots, \gamma_n \rangle$ is denoted as $|\langle g \rangle|$ and defined by: [5]

$$\left|\langle g \rangle\right| = \sqrt{\sum_{i=1..n} \gamma_i^2} \quad \textbf{(4)}$$

Because QFD teams generally prefer scaling importance in the range 0…5, we normalize our profiles to vectors of length 5 using the formula

$$\frac{\langle g \rangle * 5}{\left|\langle g \rangle\right|} = \left\langle \frac{\gamma_1 * 5}{\left|\langle g \rangle\right|}, \ldots, \frac{\gamma_n * 5}{\left|\langle g \rangle\right|} \right\rangle \quad \textbf{(5)}$$

With this normalization (5) done, we can compare two profiles based on their direction in the vector space only, because all these vectors have length 5. Two profiles pointing in the same direction represent the same knowledge about the process and its results.

The requirement profiles, as normalized vectors, show the direction our project has to go on our quest for knowledge acquisition. Furthermore, it is possible to eliminate requirements that are not contributing to the desired effect. If the weight of some requirement becomes zero, we may not need that and save all cost related to it. If our solution approach does not point into the right direction, we must change our selection of solution requirements. This simply means, find a better solution.

2.5 The Convergence Factor – Improve the Cause/Effect Relationship

We need a metric to measure how well our choice of solution profile matches the goal. This metric we call the *convergence factor*. It is the length of the profile difference between goal profile and effective profile, divided by the number of profile coefficients.

Let $\langle g \rangle = \langle \gamma_1, \ldots, \gamma_n \rangle$ be the goal profile, let $\langle x \rangle = \langle \xi_1, \ldots, \xi_m \rangle$ be the solution profile and $\varphi(\langle x \rangle) = \langle \varphi_1, \ldots, \varphi_n \rangle$ be the effective profile. Then the *convergence factor* is the square root of the length of its profile difference $\langle g \rangle - \varphi(\langle x \rangle)$ divided by the number of goal coefficients n:

$$\kappa = \sqrt{\frac{\sum_{i=1..n} (\gamma_i - \varphi_i)^2}{n}} \quad \textbf{(6)}$$

(Convergence Factor)

---

5 For n = 2, this is the theorem of Pythagoras: $\left|\langle y \rangle\right| = \sqrt{\zeta_1^2 + \zeta_2^2}$.

A convergence factor (κ) of zero means complete convergence. $\kappa = 0.2$ is generally considered good; $\kappa = 0.5$ is acceptable, as the latter indicates a deviation of direction by 10%. κ greater than one indicates a significant difference between the goal profile and the profile achieved with the chosen solution approach. It is management's decision how accurate the development shall keep its original direction.

When we have a matrix with a bad convergence factor, there are two distinct solution requirements:

1. Add better requirements to the solution profile that better supports the goal profile (e.g. better fit customer's needs), until the convergence factor decreases. This is the preferred way experienced by QFD practitioner and works in a workshop setting.
2. Use the convergence factor as the minimum target for linear optimization. There are standard mathematical algorithms that reliably decrease the convergence factor by optimizing the solution profile.

Linear optimization finds a local optimum but does not detect all possible solutions. It cannot replace the search for better solution requirements. For more details regarding linear optimization with QFD, see [8].

2.6 The Three Metrics for Six Sigma for Software – Controlling Development

Based on the convergence factor, a network of knowledge combinators allows both forward setting the required profiles for the requirements, and backwards controlling whether software development met those profiles, see Fig. 3.

Let $<g>$ be the goal profile, $<x>$ the optimized solution profile and $<x_0>$ the measured solution profile. Then $\varphi(<x>)$ is the effective goal profile and $\varphi(<x_0>)$ the measured impact profile. With these three vectors we define three metrics:

- *Convergence Factor*: The difference between goal profile $<g>$ and effective goal profile $\varphi(<x>)$ shows how accurate the cause–effect – relationship is. It is a metric for the achievement of objectives under ideal conditions.
- *Effective Deviation*: The difference between goal profile $<g>$ and measured impact profile $\varphi(<x_0>)$ shows how well the target goals are effectively met. It is a metric for the achievement of objectives as perceived by others.
- *Process Maturity Gap*: the difference between effective goal profile $\varphi(<x>)$ and measured impact profile $\varphi(<x_0>)$ shows the gap between what your development processes are aiming for and what they achieved in the real world. It is a metric for the maturity of your development process.

It is possible to use more than one measurement per functional topic, and thus combine measurements and get effective deviations from different solution topics.

The convergence factor is the most important metric, as it tells whether the knowledge combinators explain the knowledge acquisition process sufficiently well. It indicates whether the knowledge combinators are good enough.

The effective deviation reports adherence to the goals set for development and depends from software metrics collected during software development.

The process maturity gap also depends from software metrics. It is a metric for process capability. It is similar, but not equal, to the CMM capability metrics [13]. In contrary to CMM, it measures the process capability on those functional topic levels that provide the business benefit. It may even complement the CMM metrics, as it transforms the continuous CMMI capability metrics into benefits for business, see [7].

2.7 The Deming Process Chain for Software Development

The knowledge combinators provide the metrics that define – using formula (6) – the statistical variance that we must expected when developing software. The Deming process chain yields metrics that let us control the deployment of requirements throughout the software development. Thus, formula (6) plays the same role as the formula (1) for such processes that do not produce statistically relevant data samples. The Six Sigma metrics for software indicate whether the development processes produce results that meet customer's expectations, or whether it went off way.

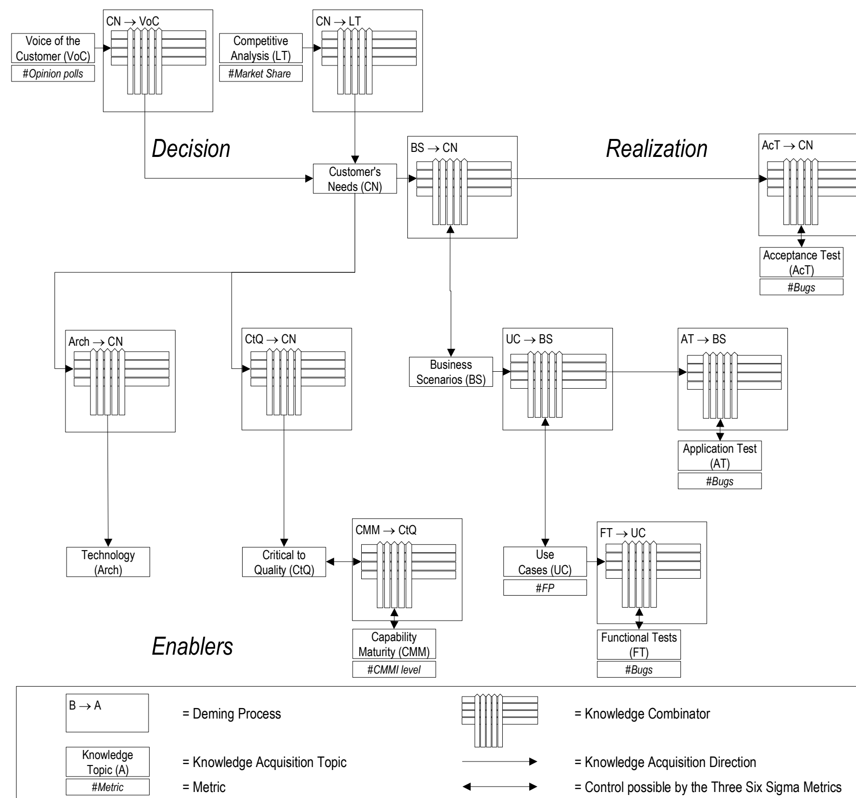

Fig. 3. Deming process chain for software development with its knowledge combinators

3 Benefits

3.1 Practical Experiences

The framework outlined in this report has evolved over time in more than 15 years. It has been extensively tested in a number of projects, and it proved to be of outstanding value. These experiences had been reported on several occasions [9].[6] However, the understanding why it worked grew only over time.

The striking correspondence between formula (1) and (6) explains these successes. What we observed in practice as a working theory is in fact based upon the same mathematical structure.

3.2 Conclusion

We conclude from the observations made in practice that the theory seems to serve well as a model for software process improvement.

Deming's generic process model is very useful to describe software development. The key insight provided by that model is that requirements for software are bound to functional topics. Experience and recent advantages in software engineering [3] demonstrate that solution requirements drive the software development processes.

The metrics derived from Akao's Quality Function Deployment provide the same benefits as statistical process control but rely on analyzing knowledge acquisition rather than statistical regression analysis. Mathematically, the metrics are very similar in structure, provided one assumes the normalization formula (4) originally proposed in [8]. For software development process engineering, this is a break-through, because it allows doing statistical process control based on cause/effect analysis rather than statistical data samples.

Although the mastery of both Six Sigma statistics and Linear Algebra is a challenge for Six Sigma Green or Black Belts, as well as for software engineering process groups, it is worth the effort because software is the most important single component for most products and services. Six Sigma will be expanded into software, and thus, mastery of software development and operational use will become decisive for the economic future of our societies.

References

1. Abran, A. et. al. (2003), COSMIC-FFP Measurement Manual - The COSMIC Implementation Guide for ISO/IEC 19761: 2003, Version 2.2, The Common Software Measurement International Consortium (COSMIC), Montréal, Kanada
2. Akao, Y. et. al. (1990), *Quality Function Deployment (QFD)*; Productivity Press, University Park, IL

[6] Experience reports are available on the web site of the author (www.e-p-o.com).

3. Beck, K. (2002), *eXtreme Programming explained*, Addison-Wesley, Boston, MA
4. Deming W. Edwards, “Out of the Crisis”, MIT Center for Advanced Engineering Study, Cambridge, Mass., 1982
5. Engeler E. (1995), *The Combinatory Programme*, Birkhäuser, Basel, Schweiz
6. Fenton, N.; Krause, P.; Neil, M. (1999), “A Probabilistic Model for Software Defect Prediction”, IEEE Transactions on Software Engineering, New York, NY
7. Fehlmann, Th. (2003), “Strategic Management by Business Metrics: An Application of Combinatory Metrics”, *International Journal of Quality & Reliability Management*, Vol. 20 No. 1, Emerald, Bradford, UK
8. Fehlmann, Th. (2004), “The Impact of Linear Algebra on QFD”, in: *International Journal of Quality & Reliability Management*, Vol. 21 No. 9, Emerald, Bradford, UK
9. Fehlmann, Th. (2005), *Six Sigma in der SW–Entwicklung,* Vieweg–Verlag, Braunschweig–Wiesbaden
10. Herzwurm G., Mellis, W., Schockert, S. (1997): *Qualitätssoftware durch Kundenorientierung*. Die Methode Quality Function Deployment (QFD). Grundlagen, Praxisleitfaden, SAP R/3 Fallbeispiel.
11. Humphrey, W.S. (1989), *Managing the Software Process*, Addison-Wesley, Boston, MA
12. International Function Points User Group IFPUG (2004), IFPUG Function Point Counting Practices Manual, Release 4.2, Princeton, NJ
13. Kulpa, M., Johnson, K. (2003), *Interpreting the CMMI*, Auerbach Publications, CRC Press, Boca Raton, FL
14. Lauesen, S. (2002), *Software Requirements: Styles and Techniques*, Addison-Wesley, Boston, MA
15. Mizuno, Sh. ed. (1988), “The 7 New QC Tools”, in: *Management for Quality Improvement*, Productivity Press, University Park, IL
16. Mizuno, Sh. and Akao, Y. ed. (1994), *QFD: The Customer-Driven Approach to Quality Planning and Deployment*, translated by Glenn Mazur, Tokyo: Asian Productivity Organization, Tokyo, Japan
17. Oudrhiri, R. (2005): “Six Sigma and DFSS for IT and Software Engineering”, in: TickIT International, 2Q05, TickIT Office, London, UK, ISSN 1354-5884
18. Pande, P.S., Neuman, R.P., Cavanagh, R.R. (2002), *The Six Sigma Way – Team Fieldbook,* McGraw-Hill, New York, NY
19. Paulk, M.C., Curtis, B., Chrissis, M.B., and Weber, Ch. V. (1993), *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute, CMU/SEI-93-TR-24, DTIC Number ADA263403, Carnegie-Mellon University, Pittsburg, PA
20. Rico, D.F. (2004), ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers, J. Ross Publishing, Boca Raton, FL.
21. Saaty, Th. (1999), „Fundamentals of the Analytic Network Process”, in: ISAHP 1999, Kobe, Japan
22. Töpfer, A. ed. (2004), *Six Sigma – Konzepte und Erfolgsbeispiele für praktizierte Null–Fehler Qualität*, 3. Auflage, Springer Verlag Berlin Heidelberg New York
23. Zultner, R. E. (1992), “Quality Function Deployment (QFD) for Software: Structured Requirements Exploration”, in: Schulmeyer/McManus (ed.): Handbook of Software Quality Assurance, 2nd Ed., Zürich 1992, S. 297-319