Defect Density Prediction with Six Sigma

Thomas Fehlmann

Abstract

Can we predict defect density in advance for software that's going into production? – The answer is "yes", if you apply statistical methods to requirements and run measurement programs for functional size and defect cost. The Six Sigma toolbox provides statistical methods for building such prediction models.

This paper explains the basics of these statistical methods without going into the details of Design for Six Sigma (DfSS) and Quality Function Deployment (QFD).

1. Introduction

Tester would love to know how many defects remain undetected when they deliver software to a customer or user. For capacity planning and in service management, knowing in advance how many people will be needed for application support would be welcome.

Intuitively it seems impossible to predict the unknown future and how many defects customers and users are able to detect – and consequently have to remove – when starting to use the new application. However, statistical methods exist for predicting the probability of finding defects by calculating the expected defect density in requirements. It works similar to weather forecast, where predicting humidity levels and temperature based on measurable physical entities leads to rain or snowfall forecasts.

We apply statistical methods to requirements and semantics of language statements rather than to physical entities. Statistical methods for requirements engineering are well known in Quality Function Deployment (QFD). While the models are mathematically strict and exact, the application scope is not. Few standards exist for measuring requirements; the best we know are the software sizing standards that meet the ISO/IEC 19761 International Standard on Functional Size Measurement. We'll focus on two: ISO/IEC 20926, and ISO/IEC 19761. These international standards are better known under their native names of IFPUG Function Points V4.1, unadjusted, and COSMIC Full Function Points V3.0.

The prediction model uses a multidimensional vector space representing requirements to the software product with a topology that describes defect density during the various stages of software development. In mathematics, this structure is called a "Manifold". The defect density function changes from project inception stage to requirements definition stage, to design and implementation stage. These stages correspond to different views on the software requirements definition and implementation process.

2. What are defects in Software?

2.1. A mistake, a bug, or a defect?

"Defects, as defined by software developers, are variances from a desired attribute. These attributes include complete and correct requirements and specifications as drawn from the desires of potential customers. Thus, defects cause software to fail to meet requirements and make customers unhappy.

And when a defect gets through during the development process, the earlier it is diagnosed, the easier and cheaper is the rectification of the defect. The end result in prevention or early detection is a product with zero or minimal defects." This statement of belief in the Six Sigma approach is expressed in [1].

The English language provides three different notions depending from where nonconformity originates:

- Developers find “Mistakes” and eliminate them;
- Testers find “Bugs”, and Developers eliminate them;
- Customers find “Defects”, Supporters assess them, Developers eliminate them, and Testers repeat testing.

With that many roles and people involved, the cost of rectification of the nonconformity raises quite a bit. Thus it is desirable if developers find all mistakes, before they become bugs, or defects at the end.

Although this is a very good idea, reality is that developers typically do not know enough to recognize mistakes before mistakes become defects – even if they *had* the time to do it.

2.2. Software development is knowledge acquisition about requirements

Software development is unlike civil engineering – you cannot expect building according a detailed plan; not even by combining best practices. Most software projects consist of ongoing knowledge acquisition – both in depth as in scope. Developers and users – both are involved into knowledge acquisition. Thus, requirements are not a static entity. Requirements change: from business needs to professionally stated requirements, then to technical requirements, architecture requirements and required features.

Unfortunately, requirements change also for other reasons: because they were stated badly, or not understood, or simply not stated at all. These kind of defects affect software development badly. For coding errors, even for logical mistakes, tools exist to detect them; missed requirements – and missing requirements – are much harder to predict.

2.3. Missed Requirements and Missing Requirements

In Six Sigma for Software, we distinguish two kinds of defects:

- A-Defects; missing or incomplete requirements, or badly stated requirements;
- B-Defects, missed requirements that had been correctly stated but not understood by developers, or not implemented. [2]

A-Defects can cause a software product being rejected in an organisation, or not used at all, or becoming a commercial failure. A-Defects are difficult to find, but they have a natural priority weighting. Some A-Defects are less important than others.

B-Defects, in turn, are less difficult to identify: you can compare the software product with the correctly specified requirements in order to identify gaps. Throughout testing will do. However, B-Defects often have no “natural” priority assigned that reflects customer’s needs. This may lead to ineffective testing or to unacceptable delays.

Defect prediction works for both types of defects the same, if B-Defects have a priority weighting similar to A-Defects.

2.4. Origins of defects

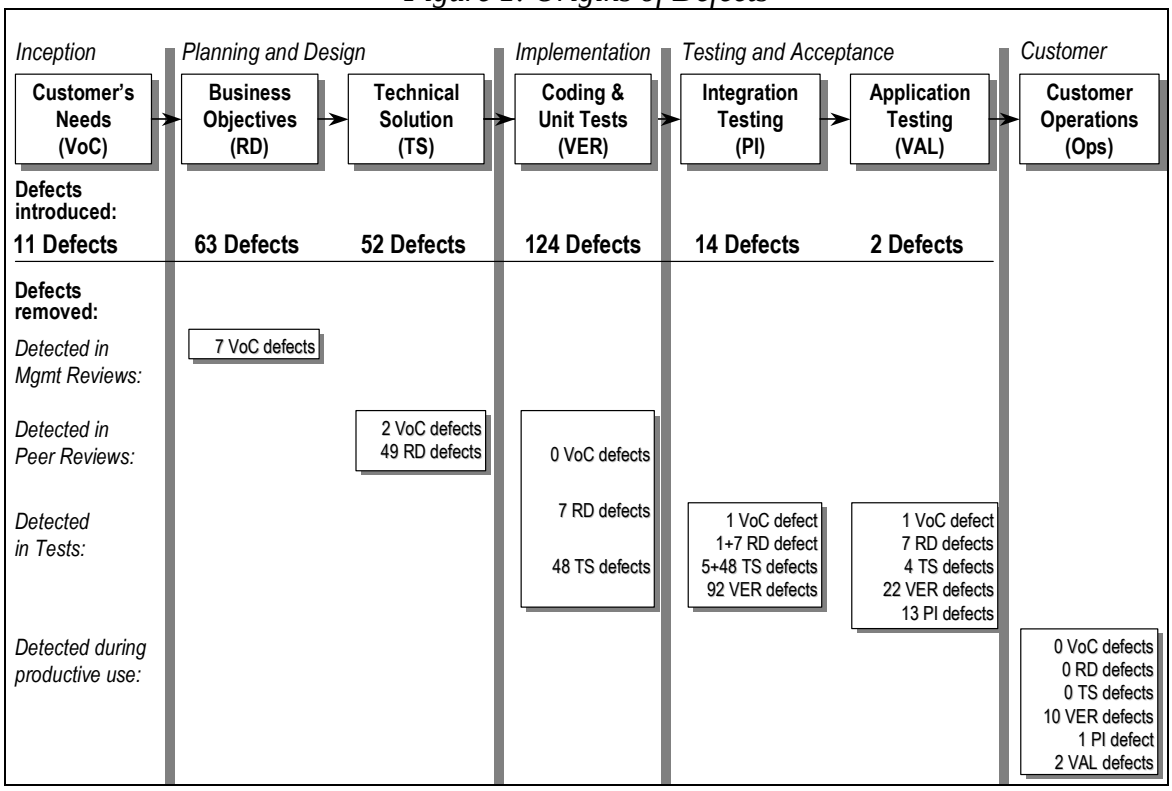
We concentrate on the following processes (in parenthesis: references to CMMI V1.2) [3]:

- Voice of the Customer (VoC, see Quality Function Deployment)
- Requirements Elicitation of Business Objectives (RD, Requirements Development)
- Architecture and Design (TS, Technical Solution)
- Coding and Unit Testing (VER, Verification)
- Integration Testing (PI, Product Integration)
- Application Testing (VAL, Validation)
- Operations (Ops, see following note on IT Service Management)

A-Defects originate typically from VoC or RD; B-Defects from TS, PI, or from VER and VAL. We assume that processes such as REQM (Requirements Engineering) or CM (Configuration Management) do not produce significant defects, because they can be fully automated. Also, we exclude impact from SAM (Supplier Agreement Management), RSKM (Risk Management), and from the project management and organisational process areas.

On the other hand, we also look at the Voice of the Customer Process that precedes Requirement Elicitation (RD), and the IT Service Management processes that collect issues and identifies software problems. This class of processes – which are detailed in ITIL [4] – is summarized as "Ops", for Operations. The only output of the process that is of interest for this scope are defects found during operations of the software product.

Figure 1: Origins of Defects



The display of Figure 1 shows nonconformities that leave the process where they were created in. Mistakes and bugs do not show up. In each process, defects are introduced and propagated to the next process. Sample defect numbers for such numbers are shown on the second line below the process names. Quality assurance is able to detect some defects, which are consequently removed; other defects are not detected and therefore passed to the next process. The numbers of defects removed – together with their origin – are displayed in the "Defects removed" area.

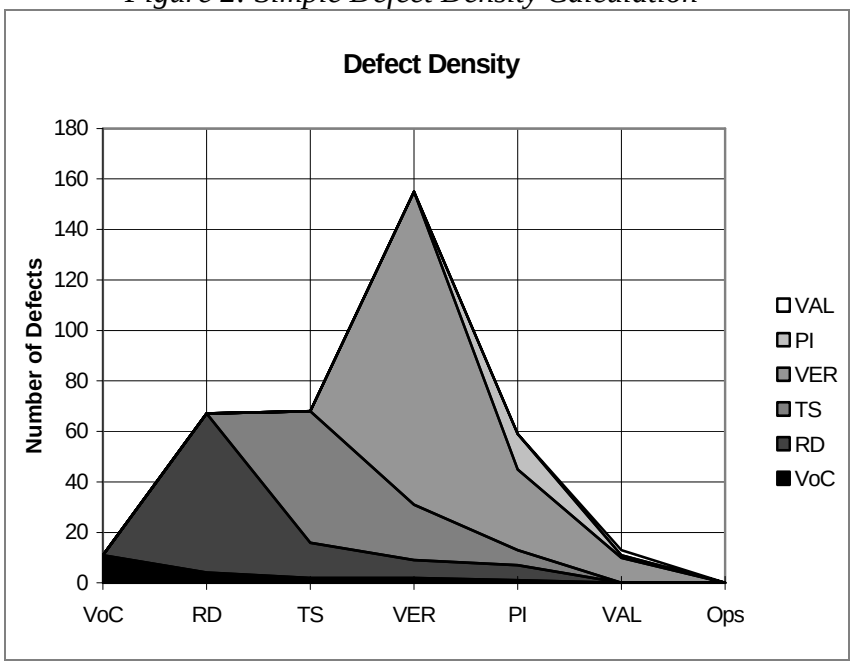
From the defects introduced during the software development processes and defect removal, we can calculate defect density as shown in Figure 2.

2.5. The Problem of Measuring Defects

Counting errors as shown in Figure 1 and Figure 2 are indicative for selecting a testing strategy, and adopting the best tactics for some software development project, but they do not allow defect density prediction, unless all defects are equally sized. This assumption is wrong, as shown in section 2.3. For measuring and predicting density, we need sizing, both sizing of software and sizing of defects.

Figure 2: Simple Defect Density Calculation

Defect Density Graph – assuming all defects are equally sized



(Numbers according Figure 1)

3. Sizing and Measuring Defects

3.1. Sizing of Business Requirements

The IFPUG Functional Size (ISO/IEC 20926:2003) [10] is a good choice for understanding and sizing user and business requirements. An IFPUG count identifies input, logical data, and output expected from the software product, and reflects user's view and business needs.

The sizing unit for Business Requirements are IFPUG Unadjusted Function Points. (UFP = Unadjusted FP).

3.2. Sizing of Technical Requirements

It is possible to count business requirements with the ISO/IEC 19761:2003 (COSMIC FFP) approach [9] as well, if the count is executed adopting a user viewpoint; however, this standard needs more in-depth analysis of the Use Cases.

For counting requirements such as solution architecture or quality criteria according ISO/IEC 9126 (modularity, extensibility, maintainability, performance, encapsulation, etc.), the IFPUG standard is less helpful. For some cases, sizing of a software solution only makes sense when adopting a suitable COSMIC FFP viewpoint; the IFPUG would probably return a size of near to zero when no persistent data exists, such as for instance a converter between different data standards, or formats.

The COSMIC FFP approach offers big advantages when sizing architectural decisions and the solution architecture since it provides a simple-to-use framework for functional sizing based on a suitable viewpoint. However, you can combine both sizing methods for defect prediction; it matters only that business and technical viewpoints are identified correctly, and that counts reflect the difference.

3.3. Identifying defects

Counting the number of defects that made their way into some "bug list" is not a suitable measurement method. First of all, it is very difficult to distinguish one defect from another. Especially for A-Defects, a missing requirement may show up several times under different

premises. For instance, if the database model lacks an entity, or a relation between entities, testers might observe different bugs that are difficult to consolidate.

On the other hand, testers sometimes observe several bugs at once but interpret it as evidence for one bug only – not being familiar with the code. Fixing this bug may reveal other bugs – or hide them. Thus, the bug count is misleading and cannot serve as a base for defect prediction.

3.4. Measuring defects as Learning Opportunities

We can avoid counting defects if we ask for the effort needed for fixing defects rather than for a count. Since we consider both A-Defects and B-Defects, we don't distinguish between Bug Fixes and Change Requests. The latter are as well indicative for a lack of our processes to identify appropriate business or technical requirements, as the former are indicative for lack of validation and verification (VER and VAL) process capability.

In order to avoid the insipid after touch that "Bug Fixing Effort" means for developers, we prefer the term "Learning Opportunities". We don't want to miss learning opportunities by biased reporting, and we have no other means to distinguish effort spent on defects from effort spent on development than by asking the developers what they are doing.

For defect density, we measure effort in terms of Person Days (PD). By dividing effort spend on Learning Opportunities by the total effort spent on some work package we get the *Learning Opportunities Ratio* (LeOR) metrics that factors skills level away.

Under the assumption that work effort depends linearly from functional size, the LeOR is the percentage of the Functional Size that is affected by bug fixes and change request implementations. Thus, functional sizing provides metrics for measuring defect density. We need neither the issue list nor requirements management (REQM) for such measurement. This makes it possible to effectively predict defect density.

3.5. LeOR and the Sigma Scale

If you cannot get time sheet reports of reasonable quality, the Sigma scale is of practical help to measure LeOR. Most often you simply can ask developers for the Sigma level they are experiencing with their work, and the result is good enough for defect prediction.

The Sigma scale translates as follows into the LeOR: e.g., for Sigma = 2.0, the success rate is 69.1%, thus the LeOR is 100% - 69.1% = 30.9%. Experienced developers can easily identify Sigma values for LeOR between 0.0 and 3.5. Sigma 0.0 means total rework; 0.5 partial rework; a Sigma of 1.0 or 1.5 is indicative for refactoring; values between 2.0 and 3.0 stand for typically development and bug-fixing cycles without (major) issues, and above 3.0 means it was right the first time. Values above Sigma 4.0 don't matter much for defects prediction.

4. Transfer Function between Different Views

4.1. A Formal Structure for Defining Knowledge Acquisition

Let's look again at the CMMI processes listed in section 2.4. For forecasting defects, the problem is that requirements – and consequently A-Defects and B-Defects – depend from the process area where they belong. You cannot say hat a business requirement, belonging to process area RD, is violated if TS is missing some particular solution requirement. The missing solution requirement may *cause* a violation of that business requirement; however, whether the particular missing TS requirement actually causes a violation, or many, or none at all, this is the question that needs to be addressed when predicting defect density. This

distinction is one of the big challenges for software engineering and software project management. Software engineers do not necessarily share business managers' viewpoints.

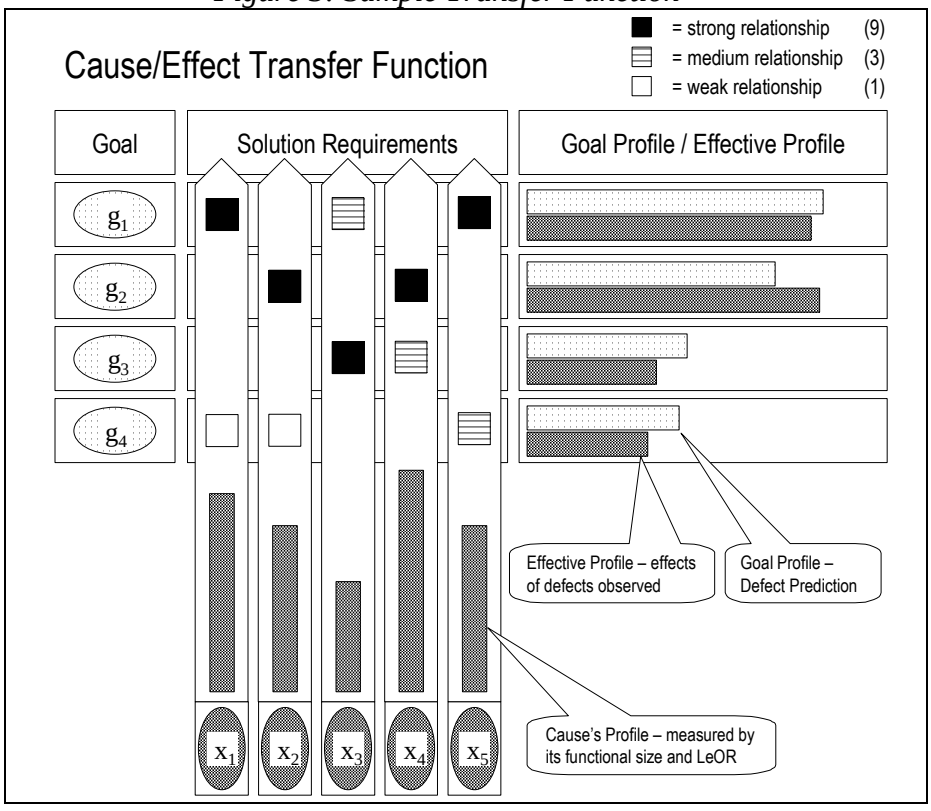
This is why the Six Sigma Transfer Function plays a central role for defect prediction. A Transfer Function for Software Development transforms requirements from one process area, i.e., Requirements Development (RD), into requirements in some other process area, for instance Technical Solution (TS), or Product Integration (PI). Six Sigma Transfer Functions for software development are well known from the Quality Function Deployment Practice, used in Design for Six Sigma (DfSS), see [2] and [7].

4.2. Defining Transfer Functions

Transfer Functions work on Knowledge Acquisition Spaces, which we explained in [5]. Requirements for the various process areas are expressed by formal knowledge about that process area's requirements. Such formal knowledge we measure as *Profiles*. Profiles are a normalised valuation of requirements, representing their relative priorities. A Transfer Function transforms a priority profile from business views into technical views. Thus a Transfer Function answers the question which technical requirements affect which business needs.

The logical structure for defining knowledge acquisition can be described using statistical methods. For additional information about using statistical methods for requirements engineering, we refer to [6].

Figure 3: Sample Transfer Function



4.3. A Sample Transfer Function

Let x be the cause's profile and y the effects' profile. Then the transfer function that maps causes into effects is $\mathbf{y} = \mathbf{T}(\mathbf{x})$. As an example, let **y** be the desired priority profile for the business needs, and **x** be the requirements profile for the Use Cases. The transfer function **T** maps the use cases to those business needs that they fulfil. Such mappings can be represented

as linear matrices, therefore suggesting that transfer functions are actually linear mappings between the vector space involved. Given the technical requirements profile $\mathbf{x} = <\xi_1,\ldots,\xi_n>$, response profile to business needs is $\mathbf{y} = \mathbf{T}(\mathbf{x}) = <\varphi_1(\mathbf{x}), \ldots, \varphi_m(\mathbf{x})>$ where $\varphi_i(\mathbf{x})$ represent the components of the vector profile $\mathbf{y} = \mathbf{T}(\mathbf{x})$. The technical requirements profile $\mathbf{x} = <\xi_1,\ldots,\xi_n>$ is knows as the *Critical Parameters* in the Six Sigma practice.

$\mathbf{T}^{-1}$ is the inverse Transfer Function. $\mathbf{T}^{-1}$ predicts the solution x that yields $\mathbf{y} = \mathbf{T}(\mathbf{x})$, given the goal profile $\mathbf{y}$. In this case, $\mathbf{x} = \mathbf{T}^{-1}(\mathbf{y})$ is the prediction. For a matrix representation of $\mathbf{T}$, $\mathbf{T}^T$ is the transpose of the transfer function. $\mathbf{T}^T$ approximates $\mathbf{T}^{-1}$ if $\mathbf{x}$ is an *Eigenvector* of $\mathbf{T}$.

4.4. Eigenvector of a Transfer Function T

An Eigenvector is a solution of the equation $[\mathbf{T}^T \bullet \mathbf{T}](x) = \lambda x$. λ a real number; usually set to $\lambda = 1$. Note that $[\mathbf{T}^T \bullet \mathbf{T}]$ needs not to be the identity function, which means, cause/effect cannot be reversed!

We need to know how good the solution $\mathbf{x}$ is: $\| [\mathbf{T}^T \bullet \mathbf{T}](x) - \lambda x \|$ is called the *Convergence Gap.* A small Convergence Gap means $\mathbf{T}^T$ approximates $\mathbf{T}^{-1}$ pretty good, or in other words: $\mathbf{x}' = \mathbf{T}^T(\mathbf{y})$ is a good prediction for $\mathbf{x} = \mathbf{T}^{-1}(\mathbf{y})$.

4.5. The Impact of Linearity

The impact of this “linearity” statement is quite important and has been discussed in many research papers. For instance, we refer to Norman Fenton’s well-known Bayesian Networks that he uses for defect prediction, see [8]. The difference to the Six Sigma approach is that it does not refer to transfer functions between different requirement spaces, it rather addresses a software engineering model framework representing the different “modules” of a software product for gathering data needed for defect prediction. Our approach has the advantage that defect prediction starts with the first early and quick functional sizing, it also has a disadvantage: statistical analysis alone will not do, you need to understand and identify the transfer functions from business to implementation that impact your software development project.

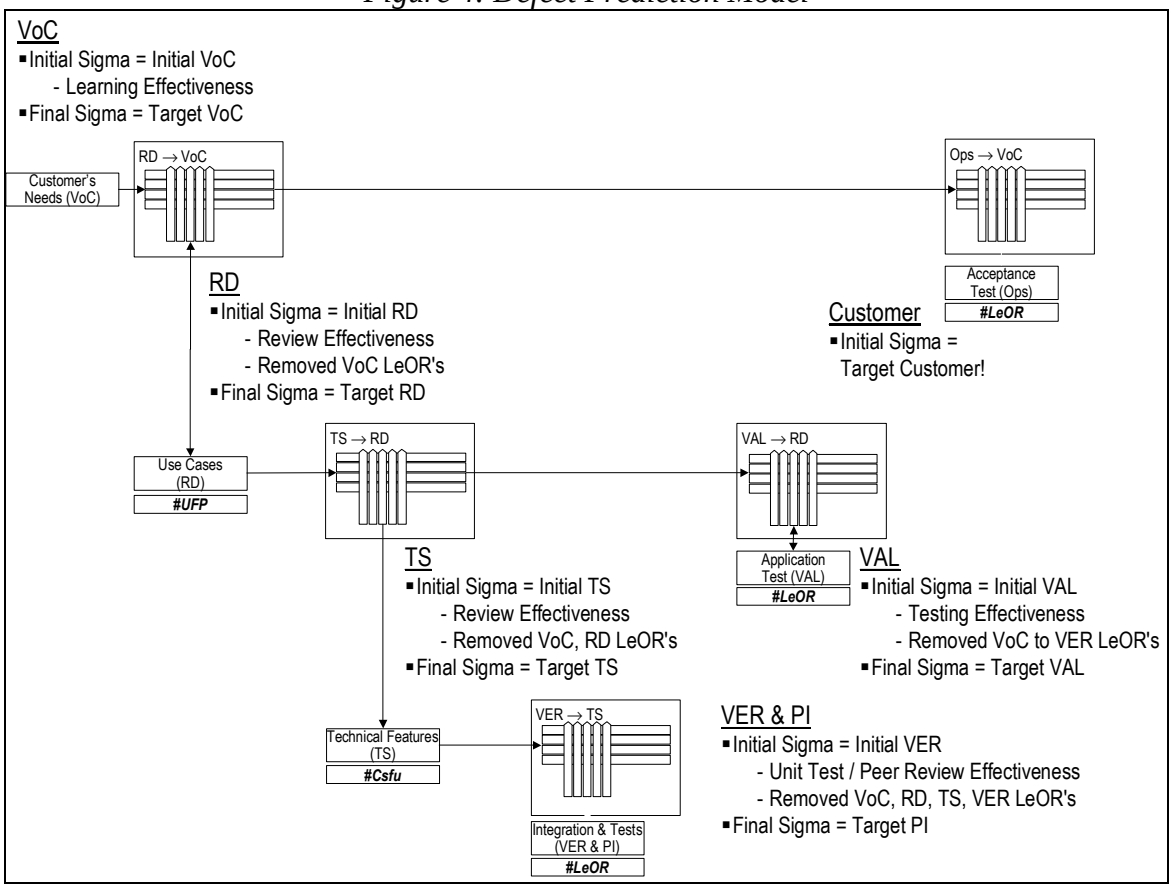
5. The Defects Prediction Model

5.1. Setting up the model

Now we have all in place to effectively build the Defects Prediction Model. The model is an excerpt from the Deming Process Chain for Software Development presented in [5] and [6], with the difference, that the Transfer Functions are not only used to map requirements from one process area onto the other, but also to map the corresponding defect densities from one area into another.

The model is build by connecting the defect density profile from the initial VoC process area to the RD process area and then to VER & PI, which stand for the implementation process areas. The Transfer Functions tell us the size of the requirements on the levels VoC, RD, TS. The VER and the VAL testing, and finally customer operations (Ops) experiences, provide the actual LeORs that allow calculating the defect density for each level. Once the VoC profile is validated, we can work on Business Requirements (RD) and Technical Solutions (TS) until we get small Convergence Gaps, i.e., Eigenvectors for our profiles. If the Convergence Gaps of our Transfer Functions are small, then our Transfer Functions are almost linear in the neighbourhood of our profiles, and we can predict defect density from the beginning, i.e., from the first functional sizing onwards.

Figure 4: Defect Prediction Model



5.2. Calibrating the Model

Our model deals with profiles. Profiles work well with transfer functions; however, a profile describes the relative distribution of defects, and not the absolute defect numbers we need to calculate defect density.

To get defect density from the profiles, we have two options:

- Use historical data for defect removal efficiency and review & test effectiveness;
- Use historical data from previous projects with similar scope in the same market.

Calibration of the model can occur at every level; for instance, using measurements at the TS / VER level.

5.3. The Starting Point

The LeOR for VoC we can also infer from previous experiences and their final success. It tells us, how many A-Defects we had to remove before getting successful feedback from market or customer. The Sigma scale presented in section 3.5 is applicable for marketers as well. If everything fails, we rely on the assumption that our initial LeOR is Sigma = 1.0.

6. Conclusions

Although defect prediction is not simple, it is easily adoptable for an organization that regularly uses Design for Six Sigma (DfSS) – or Quality Function Deployment – to control their software development. Sensitivity analysis proves usefulness of the model even with minimum calibration. The better calibration data is, the more reliable becomes prediction.

Experience reports have not been published so far; however, calculation tools and examples are available.

7. References

[1] Soni, M. (2009), "Defect Prevention: Reducing Costs and Enhancing Quality", published on http://software.isixsigma.com/, online in March 2009

[2] Fehlmann, Th. (2005), "Six Sigma in der SW–Entwicklung", Vieweg–Verlag, Braunschweig–Wiesbaden

[3] "CMMI® for Development, Version 1.2", Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, August 2006.

[4] "IT Infrastructure Library (ITIL®) V3", published on http://www.itil.org/, June 2007

[5] Fehlmann ,Th. (2006), "Statistical Process Control for Software Development – Six Sigma for Software revisited", in: EuroSPI 2006 Industrial Proceedings, pp. 10.15, Joensuu, Finland

[6] Fehlmann ,Th., Santillo L. (2007), "Defect Density Prediction with Six Sigma", in: SMEF Roma 2007, Roma, Italy

[7] Fehlmann, Th. (2004), "The Impact of Linear Algebra on QFD", in: International Journal of Quality & Reliability Management, Vol. 21 No. 9, Emerald, Bradford, UK

[8] Fenton, N., Krause, P., Neil, M. (1999), "A Probabilistic Model for Software Defect Prediction", IEEE Transactions on Software Engineering, New York, NY

[9] Abran, A. e.a.,"COSMIC FFP Measurement Manual 3.0", 2007, www.lrgl.uqam.ca/cosmic-fpp.

[10] International Function Points User Group IFPUG (2004), "IFPUG Function Point Counting Practices Manual", Release 4.2, Princeton, NJ