Legacy System Exorcism by Pareto's Principle

Kristoffer Kvam
Telenor Nordic, IT, CRM
Snaroyaveien 30, 1331
Fornebu Oslo, Norway

Rodin Lie
Telenor Nordic, IT, CRM
Snaroyaveien 30, 1331
Fornebu Oslo, Norway

Daniel Bakkelund
Telenor Nordic, IT, CRM
Snaroyaveien 30, 1331
Fornebu Oslo, Norway

kristoffer.kvam@telenor.com, rodin.lie@telenor.com daniel.bakkelund@telenor.com

ABSTRACT

Exorcism is mainly thought of as the rite of driving out the Devil and his demons from possessed persons. This text is about the same process except here the target is a legacy software system. The target system was a major component based system having been developed over 7 years by 30 to 60 people continuously under a classic plan driven approach. The Pareto Principle, or 80/20 rule as it is often called, is used as the framework to prioritize activities in a major reengineering initiative on the system from limited resources. The initiative's main focus was to increase the developer productivity in the maintenance project in the system by 25 percent. Typical agile practices were the inspiration for many of the changes implemented through the project.

A measurement program is presented for validating success, and the XRadar open source tool is used for measuring the program. In one year, the productivity increase was above 30 percent. There seems to be a high correlation between productivity and the implementation of the agile practices such as short iterations, daily standup-meetings and pair programming as substitutes with the practice of a formal QA regime. During the same period the error proneness of the system decreased with several magnitudes and our definition of the internal software quality increased by 22 percent. Hence, based on our measurements, the increased productivity was not substituted by lower quality in the system - on the contrary.

Keywords: Legacy System, Reengineering, Refactoring, Agile Development, Software Metrics, Open Source, Pareto Principle

1. INTRODUCTION

1.1 The Principle

In 1906, Italian economist and sociologist, Vilfredo Pareto created a mathematical formula to describe the uneven income distribution in Switzerland at that time, observing that eighty percent of the wealth was held by a mere twenty percent of the families. Further empirical studies for other time periods, for other countries, produced the stunning result that they all followed the same pattern. This was the inspiration of the Pareto Principle - often called the 80/20 rule [5].

Since then this law has been applied in a wide variety of domains to prioritize activities and model distributions. The principle is used within areas such as business and management, economics, marketing, networking and software. Some interesting examples of occurrences are:

- 80 percent of your deliveries comes from 20 percent of your time [9].
- 80 percent of your profits come from 20 percent of your customers [9].
- 80 percent of process defects arise from 20 percent of the process issues [5].

1.2 The Principle Applied to Software Reengineering

Mature software systems have a mysterious tendency to produce highly unexpected errors and maintenance is a pain. Taking inspiration from the Second Law of Thermodynamics some call this phenomenon increasing system entropy: In time a system experiences increasing disorder if not explicitly tended to [4]. This disorder adds accidental complexity to the problem domain's inherent complexity, something many organisations experience as their systems mature.

This paper is an experience report of using the Pareto Principle within the domain of software system reengineering and process transformation to agile practices. Specifically we show how we used the principle to prioritize what to do in order to increase development productivity and the quality of a large legacy system. A framework for measuring key business performance indicators is used to evaluate whether the approach was successful or not. We chose to call the whole process Legacy System Exorcism.

We will first present the context that the exorcism was conducted under. Then we will present the measurement program that was build to validate success and prevent a return to old sinns after the project was finished. Following that the changes performed by the reengineering project will be presented. We end will the quantifiable results and a discussion on the project's success.

2. THE CONTEXT

2.1 The System

Telenor is Norway's largest telecommunications company with numerous international interests.

Copyright is held by the author/owner.
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

COS is a middleware system, serving the mobile division in Telenor, designed to give front-end applications a consistent view across multiple back-end systems. There are more than 20 front end applications serving retail outlets, customer support, large corporate customers and internal functions. The back end systems include Sybase and Oracle databases, network connections and mainframes, all of which are logically interconnected through the use of batch jobs, scripts and database stored procedures. Some core metrics of the system:

- 30 to 60 developers at any time.
- More than 40 client systems and 20 backend systems.
- 130K Non Commented Source Statements (NCSS). 1000K generated NCSS.
- Exceeding 7 Million Transactions per day.

COS had evolved over five years into a large system, composed of many subsystems. Development went though stages with QA between each stage - a typical plan driven approach. Most developers supporting the system had above 5 years of programming experience and master level education. There was also a high majority of motivated new employees in the system with an interest in change. The composition of employees and consultants varied through the systems development, but on average there had been an overweight of consultants doing development work in the system.

After a five year period of sustained development and low focus on maintenance, the problems were manifold [6]. The Pareto project was instigated with solving the problems.

2.2 The Project

A project proposal that sounds like "saving the system from the entropy spiral" will likely face scepticism from management. The project's business case must show that the increased life of the system defends the investment. Our approach to the business case was to focus on deliverables that would reduce the cost of maintenance and implementing changes in the system. The resulting project effect goals that affected these were:

1. Increased productivity in the system's maintenance organization by 25 percent.
2. Reduced error proneness in the system.
3. Measured internal system quality enhancement.

The productivity goal (1) was enough to support the business case, and hence the most important external goal of the project. Goal (2) and (3) was set for quality assurance of the changes implemented. Focus on cost targets above typical revenue increasing targets, such as time-to-delivery and increased system flexibility, was due to the internal mindset in the organisation at the time of the project. Further details on the business case are company confidential, but the break even time for the project investment was 2 years. The project lasted for one year, and the success was to be measured a half year after the project delivery.

3. MEASUREMENT PROGRAMME

The measurement programme was implemented as a part of the Pareto project. The literature is rich on software measurement, and we relied on established references when building our own programme. We used the GQM (Goal-Question-Metric) [1] paradigm for defining our measurements. The inspiration for the measurement definitions and their classification are based on [2] and references within. The inspiration for developing a measurement breakdown under internal quality were based on methods described in [10].

3.1 XRadar

The XRadar [11] was built to solve the metric monitoring-needs of the system. The tool implements the internal quality measure as mentioned above as an easily configurable metric for other systems. The tool is also under way to be enhanced with the productivity and error proneness measures as well. A central design requirement for the XRadar was that all stake holders should be offered their preferred view of the system: managers, architects and developers. Hence, you can navigate from abstract system representations through modules, packages, classes, source code and documentation (javadoc) - everything integrated. Views consist of graphs, tables and reports on all levels.

The XRadar offers two ways of viewing the system: statics and dynamics. XRadar Statics reports on the current build of the system. XRadar Dynamics includes the time dimension and presents historical trends with respect to different metrics, and hence displays how the system has evolved over time. This is important: imagine a stock analyst giving investment advice based only on the current stock prices with no heed to the history of the stock? We believe investing in software system is no different.

After the Pareto project the XRadar was made free and open source under a BSD licence. The XRadar now has an active user community that contributes to the tool in several ways. The user community consists of both small and large organisations. Several consulting firms also use it for analysis of their client's software as well as quality control on their deliverables to their clients. See [11] and [7] for further details such as report examples, tool use and the architecture of the XRadar.

3.2 Development Productivity

Measuring development productivity is one of the most controversial issues within the software metrics field. The metrics have their strengths and weaknesses, but they all have a common theme. The main idea is that productivity is a measure of some unit of delivery per resource.

Classic units of delivery are lines of code and function points, but we did not feel that either met our requirements to relevance or ease of collection. All changes to the system are done through change requests in the integrated source control and configuration management system. Hence, we chose the change request (CR) as the unit of delivery:

$$Productivity = \frac{ChangeRequests}{Day * Resources} \quad (1)$$

The division by resources was made because the number of developers involved in maintenance changes a lot in the organisation. There is no doubt that this measure has its limitations, and need proof in order to be more valid. See

the Discussion section in the last part of the paper for details on that.

3.3 System Error Proneness

When analysing defects, we had several views to consider, but we chose the simplest and most critical measure to the business. Specifically, we defined the error proneness as a measure of critical defects corrected in production versions of the system. The optimal value was naturally 0.

$$ErrorProneness = \frac{CriticalDefects}{Day * Resource} \quad (2)$$

3.4 Internal Software Quality

Internal quality has been defined as the quality of "attributes of a software product that are dependent only on the product itself" [2]. This definition is apposed to external quality which is dependent on external factors such as the users and the machine environment. The productivity and error proneness measures as defined above are examples of such external measures. Our internal quality measurement was an aggregate measure of architecture quality, code quality and test quality. The final aggregate metric as well as its sub-components are all of a value between 0 and 1. 0 is the worst value, 1 is the best.

The final quality metric, Internal Quality (IQ), is implemented as a configurable measure in the XRadar. The measure we set up was:

$$IQ = 0.35 * A + 0.3 * C + 0.35 * T \quad (3)$$

In the equation above, A is the architectural quality, C is the code quality and T is the test quality. The calculation of these sub-components (all in the range from 0 to 1) are summed up in table 1. Notice how the IQ metric is calculated by a hierarchy of sub-components.

The weights of the sub-components were set by system experts based on three factors:

1. The expected cost of increasing (making better) a particular component.
2. The expected effect of the component on the error proneness measure.
3. The expected effect of the component on the productivity measure.

A more scientific approach could have been to set the coefficients by a regression analysis. That was not done.

4. PROJECT DELIVERIES

Based on the project effect goals, the project deliveries were split into several areas. This section will describe each area in turn: what it contains, how they were prioritized using the Pareto principle and how they were believed to affect the indicators. See [6] for even more discussion on the architectural and code related deliveries.

4.1 Area: Architectural Quality

4.1.1 Problems

The system was known to be hard to maintain. Using the XRadar for inspections, we identified the following malicious symptoms with respect to architecture:

- A lot of cyclic dependencies between modules.
- Code duplications, both of "simple copy-paste"-type and the more dangerous "duplicated business logic"-type.
- Poor system-component orthogonality, causing changes in one part of the system to result in unexpected behaviour in other places.
- Inadequate layering. A great deal of back-end system detail was directly exposed in the system's API.

4.1.2 Hypothesis

Inspired by the Pareto principle, we established the following hypotheses to use in the further prioritization and selection of what to focus on.

1. *20 percent of the system functionality constitutes 80 percent of the system value.*
2. *20 percent of the system architecture problems causes 80 percent of the observed system problems.*

4.1.3 Delivery

After several rounds of discussion, everything from a rewrite to minor fixes here and there where considered. However, a rewrite would violate hypothesis number two since we would spend a lot of effort fixing code that already worked. On the other hand, we had to come up with something to tackle the 20 percent of the system causing most problems. In the end we decided on a refactoring strategy, focusing on cyclic dependencies and code duplications. The idea was that these problems to a large degree were the cause of the other two.

The deliveries of the project with respect to architecture ended up being:

- Defining a dependency graph for the system. A dependency graph is a DAG describing the legal dependencies between the different modules of the system. This asset describes both which main modules the system is meant to consist of as well as which system is allowed to access which. This induces both vertical and horizontal layers in the system. The actual compliance to the defined dependency graph was to be monitored using the XRadar both during the project and afterwards.
- Improving the system's compliance to the dependency graph. A lot of the cyclic dependencies were caused by business logic residing in too high tiers in the architecture, forcing lower layer to either make calls to higher layers or duplicate the required logic. The delivery would be removal of cyclic dependencies and duplicated business logic. This should in turn result in better modularization. This would include moving around misplaced business logic to remove cyclic dependencies and duplication of business logic as well as refining the modularization of the system.
- Build a set a frameworks that would help removing the illegal dependencies.

Metric Level 1	Metric Level 2	Formula
Architecture (A)		$A = 0.4 * M + 0.6 * C$
	Modularization(L)	$packages(illegaldependencies = true)/packages$
	Cohesion(C)	$packages(cycles = true)/packages$
Code (C)		$C = 0.15 * DOC + 0.4 * DRY + 0.3 * FRE + 0.15 * STY$
	Documentation(DOC)	$javadocs/(methods + classes)$
	Dryness(DRY)	$classes(duplications = true)/classes$
	Freshness(FRE)	$classes(codeviolations = true)/classes$
	Stylishness(STY)	$classes(styleerrors = true)/classes$
Test Suite (T)		*Unit Test Code Coverage*

Table 1: The subcomponents of the internal quality measure.

4.1.4 Expected Impact

The effects of this activity would directly affect the Architecture (A), Code (C) and Test Suite (T) parts of the internal quality definition. You would directly make changes to code and architecture which would increase the architecture metric, and refactor code while working on the modularisation. Tests would be written to secure success.

4.2 Area: Code Quality

4.2.1 Problems

Being guided by the XRadar and code read-throughs, there was no difficulty seeing that a large part of the code in the system was suffering from a lot of unnecessary complexity and low readability. There were parts which were producing worrisome measurements and were also heavily maintained. Other parts did not seem bad at all. Interviews with maintainers were producing similar conclusions, but they had all their favorite areas to complain about depending on the domains they had worked within.

4.2.2 Hypothesis

The Pareto principles affecting our choices within the code quality domain were:

1. *20 percent of the code is given 80 percent of the maintenance.*

2. *20 percent of the maintained code produces 80 percent of system defects.*

4.2.3 Delivery

If these two hypothesizes about the system were true, there was very little sense in refactoring large parts of the code in order to increase productivity or the error proneness. The investment would not pay off. Only a strict prioritization would give a payoff.

In fact, the few things that was done to the code was directly linked to the deliveries mentioned under architecture above. Little effort was put into refactoring code only due to a lot of violations. On the other hand, several important deliveries were done for the future:

- Remove and deprecate as much code as possible that was not referenced or being used.

- Monitor code quality development and prioritize bad candidates automatically with the XRadar.

4.2.4 Expected Impact

The effects of this activity were expected affect the Code Quality (C) part of the internal quality definition.

4.3 Area: Test Quality

4.3.1 Delivery

Our decisions related to test, followed naturally from the Pareto principles mentioned under the previous Code Quality section. There was no sense in developing a large test suite on code that was not maintained. The deliveries followed accordingly:

- Deliver an automatic unit test framework.

- Deliver an automatic regression (acceptance) test framework.

- Use these frameworks to create tests for all code that was changed on the project.

4.3.2 Expected Impact

These activities were necessary to be able to perform the code changes in the project. You needed the frameworks to avoid introduction of errors. It was also crucial if the practice of test-driven development could be introduced (see below for more on that).

4.4 Area: Culture and Process

4.4.1 Problems

For most actors in and outside the development organization, the system seemed like a large blob of code that mysteriously worked. A few core developers had their own special domains, without seeing the system as a whole. Little communication happened between the developers. The mood was excellent, and all were very dedicated to technology, but little was done to share knowledge and discuss the overall strategies for the future. Most architects were simply senior programmers that had got the role, and their beliefs were in many ways stuck in a past which were changing to rapidly for them to follow.

Changes in the system were coordinated by a change manager that assigned tasks to individual developers. There was little communication between the client of the changes and the developers, and requirements were sparse. The process was plan driven and waterfall-inspired. There were 4 month iterations ending with a production release and a strict prioritization process between stake holders that were striving to increase business value.

4.4.2 Hypothesis

The Pareto principles affecting our choices within the cultural and process domain were:

1. *20 percent of the developers' possess 80 percent of the system knowledge.*
2. *20 percent of the formal process elements cause 80 percent process inefficiencies.*

4.4.3 Delivery

Even though the technological changes done to the system were expected to give fundamental gains, we believed that true continuing success could only obtained by reshaping the values of the organisation and letting those values be reflected in the underlying process.

From the above hypothesizes, our conclusions were that we should try to breed a culture and a supporting process that excelled in knowledge sharing and communication with all involved parts. Our belief was that we could gain great strides in the productivity and quality focus of our organization by focusing at some limited areas. Specifically, we needed a way get rid of the QA regime that was observed to take a lot of time and effort, while not increasing error proneness and the internal quality. The following changes were made in the numbered order:

1. Introduce a new and modern configuration management system that supported agile practices such as refactoring and could map change control issues directly to the code.
2. Weekly, invite all the programmers to a half hour talk about technical and cultural issues.
3. Introduce interviews for new consultants focusing on a selection of elements from references such as [4] and [3].
4. Introduce test driven development into the development process.
5. Replace or move central programmers and architects promoting an unwanted culture.
6. Introduce the Scrum agile process [8] with techniques such as using shorter delivery iterations, more communication with stake holders and daily stand-up meetings.
7. Introduce pair programming into the development process to replace the old QA regime.
8. Start a weekly meeting place ("Scrum of Scrums" [8]) where selected programmers involved the various project teams exchange operative information.

An important factor is that all these changes were not all instigated by the Pareto project. Only change 1-3 was instigated by the project, and the other changes came later. Most of the later ideas were discussed during the project, and key project participants were driving forces when they were instigated.

4.4.4 Expected Impact

The combined effects of these changes were believed to become formidable. Changes such as test driven development, pair programming and weekly development forums were believed to increase the internal system quality measure in all its dimensions a great deal. That again was believed to increase productivity and reduce error proneness. The changes to the process such as the new configuration management system, Scrum and increased communication in the organization was mainly believed to give impact on productivity through developer focus and value chain understanding.

5. RESULTS

The results based on the success factor metrics are presented in table 2. Since the configuration management system was introduced during the project, the productivity and error proneness values are only present one year back in time.

The success criteria for the project were to be measured between release 11 (base line) and 15.1. As the table above shows, the increases in the key project goals were the following:

- Increase in productivity by more than 30 percent. The first measure (15) was 51 percent above the base line, while the second measure is 32 percent above the base line.
- Decrease in error proneness by several orders of magnitude.
- Increase in internal quality by 22 percent.

The graphs of productivity and error proneness can be found in figure 1 , while the internal quality can be found in figure 2.

6. DISCUSSION

Below, we discuss the various success goals of the Pareto project and how well they correlated with our hypothesizes.

6.1 Productivity

Assuming that the productivity measurement is correct, the initial data shows that the two and only real changes that definitely had a positive effect was the introduction of the Scrum process and the replacement of the formal QA process by pair programming. Exactly to what degree each had on the measure is difficult to say, but probably it comes from a combination of the two.

For providing more evidence on the productivity increase, we see that 2 indicators correlate well with the result:

1. Before and after the increase in the measure, our analysis has shown us that the mean size of CRs are the in terms of classes involved had the same mean (2.4 classes).
2. Before and after the increase in the measure, the tasks started were all based on the same paradigm of setting initial estimates. Before the change, the organisation

Release	Date	Productivity	Error Proneness	Internal Quality	Significant Event
7.2	01.05.2002	NA	NA	0.28	
8.1	01.11.2002	NA	NA	0.29	
9.3	01.06.2003	NA	NA	0.29	
10	01.09.2003	NA	NA	0.31	Pareto project instigated Unit test framework introduced
11	01.11.2003	NA	NA	0.33	Configuration management system introduced interview program for new consultants started
12	01.02.2004	$8.41 * 10^{-2}$	$2.02 * 10^{-3}$	0.41	
13	25.05.2004	$7.00 * 10^{-2}$	$6.00 * 10^{-3}$	0.46	Weekly knowledge sharing meeting started Pareto project ends
14	15.09.2004	$8.24 * 10^{-2}$	0.00	0.47	Test driven development required
15	17.11.2004	$12.8 * 10^{-2}$	$1.22 * 10^{-3}$	0.48	Scrum introduced Pair programming introduced
15.1	29.01.2005	$11.1 * 10^{-2}$	0.00	0.50	Scrum of Scrums introduced Major stability case of the servers drains the organisation of resources

Table 2: Changes of the metrics in the system through the releases.

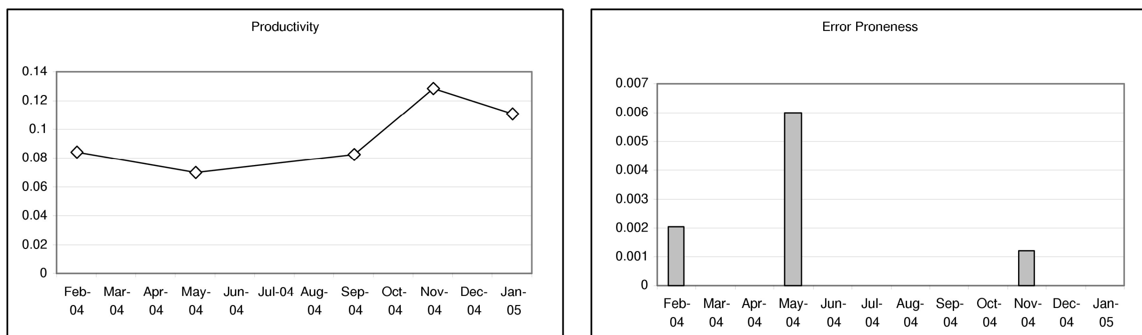

Figure 1: The productivity and error proneness.

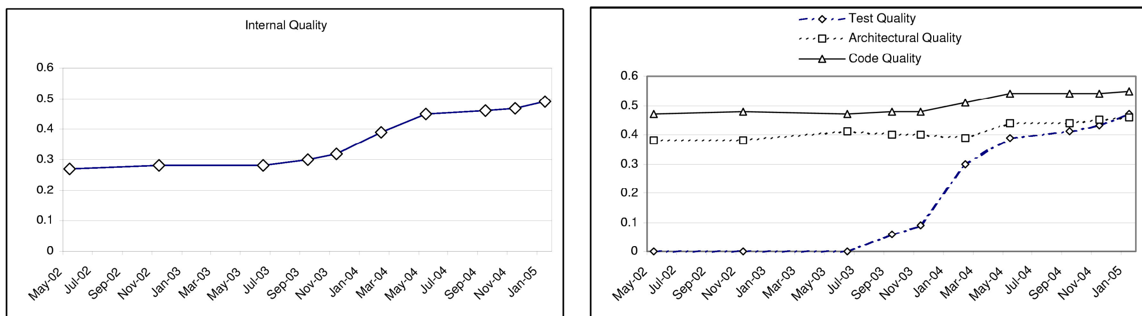

Figure 2: The internal quality measure and its components. Note that the graphs goes more than 2 and a half year back in time. This is apposed to the productivity and error proneness which only goes back 1 year.

had a delivery accuracy to these estimates of -20 percent. After the change the delivery accuracy to these estimates was +30 percent. This correlates well with a 50 percent increase in productivity.

Still, this is not enough as a proof of the measure. If we had conducted a survey of system stake holders (such as management, clients, developers) before and after the measure was done, and saw a similar trend, we would have been more comfortable.

6.2 Error Proneness

Error proneness fell a lot after the project, and can likely to be attributed to the sum of all the activities that were done. Still, we can easily identify a caveat here. As the results show, the error proneness increased dramatically shortly after the project. Those errors are likely due the reengineering project changes.

6.3 Internal Quality

First of all, there is no indication of there being correlation between our internal quality measure and the productivity measure. This correlation was one of the major hypothesizes of the project, and seems to be invalid. There is more proof that there is a correlation between this measure and decreased error proneness. Still, the validity of this measure is uncertain. There is a need for more data to be certain.

6.4 Project Success Validation

Based on the above there are two ways of looking project success. If we look at the project in retrospect, there is little proof showing that the changes done by the project actually produced the shift in productivity. As discussed, there are indications that the project reduced error proneness, but it is not conclusive.

If we, on the other hand, look at this project as knowledge and maturity creation process in the organisation, it was a definite success. After all, the defined success criteria were met within the project validation time and more than that. We believe that introducing test driven development, Scrum and pair programming would not have been possible without the changes made to the system as well as the cultural initiative that were started.

If we had done the project again, there is always the question of what should have been done different. The biggest mistake made was forgetting to early analyse quantitatively how different change alternatives evaluated by the project would impact the core success criteria. If we had done that, we could have benefited much more from prioritization based on cost/benefit.

7. CONCLUSION

This text has presented our experience in a major reengineering initiative on a legacy application. The results indicate that the work done, combined with initiatives started after the project based on its created momentum, made the project a success. A formidable increase in productivity resulted from the initiative, and the quality of the system was increased along some major dimensions, but on far from all. The belief was that there is an alternative to a total system rewrite, and based on our measurement program we were right.

The paper leaves as many questions as it answers. There is a definite need for further correlation studies and regression analyses of the data based on the COS system as well as other internal and external systems. In addition, an interesting study could be on how other external attributes of the system were affected by the changes made. Such could be measures of availability, responsiveness and system supportability. We are also in the process of conducting an experiment with one of our academic partners, Simula, on the results from the pair programming, and believe that will give further indications of the results obtained.

Entropy is a never ending force pushing our system and its development organization out of balance. New exorcists must be educated to continually initiate countermeasures agianst this entropy demon. At least now we are not left blindfolded when analysing the system and evaluating activities. We now have the advantage, thanks to the XRadar, system testability, system flexibility and not to forget the solid communication channels that are in place. In the end, our exorcists now have the ability to give the systems its death sentence. A pragmatic evaluation by the Pareto Principle might one day recommend a complete rewrite. After all, no systems live forever.

8. REFERENCES

[1] Basili et. al. A methodology for collecting valid software engineering data. *IEEE Tanssactions on Software Engineering*, 10(6):728–38, 1984.

[2] Fenton Norman E. et. al. *Software Metrics: A Rigorous and Practical Approach, Revised Edition.* Metrics and Models in Software Qulality Engineering, 1998.

[3] Fowler Martin et. al. *Refactoring: Improving the Design of Existing Code.* 1999.

[4] Hunt A. et. al. *Pragmatic Programmer, from Journeyman to Master.* 2000.

[5] Juran J.M. et. al. *Juran's Quality Control Handbook, 4th Edition.* 1998.

[6] Kvam Kristoffer et. al. Cynical reengineering. *XP2004 Procedings*, 2004.

[7] Kvam Kristoffer et. al. A tool for cynical reengineering. *rOOts 2004 Procedings*, 2004.

[8] Schwaber et. al. *Scrum.* 2000.

[9] Koch. *The 80/20 Principle: The Secret to Success by Achieving More with Less.* 1999.

[10] Gilb Tom. *Principles of Software Engineering Management.* 1999.

[11] XRadar. http://xradar.sourceforge.net/.