Software Economics: Quality-Based Return-on-Investment Model

Ljubomir Lazić
Department for Mathematics and Informatics
State University of Novi Pazar
SERBIA
llazic@np.ac.rs, http://www.np.ac.rs

Nikos E Mastorakis
Technical University of Sofia,
English Language Faculty of Engineering
Industrial Engineering, Sofia, BULGARIA
http://www.wseas.org/mastorakis

Abstract: - Along with the ever more apparent importance and criticality of software systems for modern societies, arises the urgent need to deal efficiently with the quality assurance of these systems. Even though the necessity of investments into software quality should not be underestimated, it seems economically unwise to invest seemingly random amounts of money into quality assurance. The precise prediction of the costs and benefits of various software quality assurance techniques within a particular project allows for economically sound decision-making. This article explains the return on investment rate (ROI) of Software Process Improvement (SPI), and introduces practical metrics and models for the ROI of SPI. Furthermore an analytical idealized model of defect detection techniques is presented. It provides a range of metrics: the ROTI of software quality assurance for example. The method of ROTI calculation is exemplified in this paper. In conclusion, an overview on the debate when software is purchased, concerning quality and cost ascertaining in general will be given. Although today there are a number of techniques to verify the cost-effectiveness of quality assurance, the results are thus far often unsatisfactory. More importantly, this article helps sort through the seldom and often confusing literature by identifying a small set of practical metrics, models, and examples for the ROI of SPI.

Key-Words: - Software Economics, Return on Investment, Software Process Improvement, Software Quality.

1 Introduction

Our research [4-6][1] concluded that developing software is for most organizations no longer an independent software project, but is part of a business case which includes all disciplines involved. In order to stay competitive, companies must deliver high quality products on time and within budget. Although the development cost was very important, quality, lead-time, and delivery precision were considered as the most important factors. Therefore, an evaluation of what effect the implemented concept had on these factors of Quality-Based Return-on-Investment, using Cost Benefit Analyses, was of interest. This paper satisfies these objectives by designing, constructing, and exercising a multi-part methodology consisting of a Defect Removal Model, Cost and Benefit Data, Return-on-Investment Model, Break Even Point Model, and Costs and Benefits of Alternatives, which all lead up to a Cost and Benefit Model (as shown in Fig. 1).

2 Costs and benefits of SPI strategies

Costs and benefits of Software Process Improvement (SPI) strategies will be evaluated by a variety of interrelated techniques, starting with the Defect Removal Model. The Defect Removal Model, as explained later, is a technique for evaluating SPI method effectiveness, and once economic models are factored in, provides an empirically valid approach for comparing the costs and benefits of SPI methods. Obviously, existing cost and benefit data for SPI methods selected from the Literature Survey will be judiciously factored into, and drive, each of the individual analytical models. A Return-on-Investment (ROI) Model will be designed, based on the Defect Removal Model and populated by empirical cost and benefit data, in order to arrive at quality, productivity, cost, break even, and of course, ROI estimates. Eventually, a SPI strategy Cost and Benefit Model will be constructed from Cost and Benefit Criteria, SPI Strategy Alternatives, and Cost and Benefits of Alternatives: Buy or Produce Software for company's Information System business tasks. The design of the Methodology was significantly

[1] This work was supported in part by the Ministry of Science and Technological Development of the Republic of Serbia under Grant No. TR-13018.

influenced by Kan's [2], Jones' [7], Rico's [8] and Lazic's [4-6] Defect Removal Model-based comparisons of SPI costs and benefits.

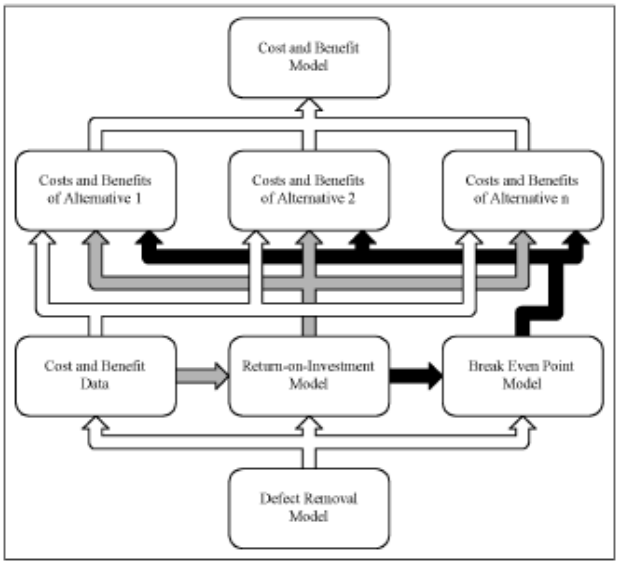

Fig. 1 Methodology for Evaluating and Selecting Costs and Benefits

An analysis of SPI costs and benefits also served as primary influence for the design of the Methodology. Their study, however, was the primary influence for two reasons, it is comprehensive in nature, and it exhibits a uniquely broad range of comparative economic analyses between SPI methods. In addition, their study stands alone in unlocking economic analyses associated with the Clean Room Methodology, Software Reuse, and even the Software Inspection Process. The study goes even further than that, in creating and establishing a valid empirically-based methodology for using existing cost and benefit data and analyses, for evaluating and selecting SPI methods.

Furthermore, Lazic's text [4-6] on SPI strategies also influenced the design and direction of the Methodology, explicitly identifying the Software Inspection Process as having an overwhelming impact on bottom line organizational performance. Thus, his works [4-6] helped justify the creation and significance of the ROI Model, which will be explained in greater detail later. Kan's and Lazic's Defect Removal Model-based SPI method comparison, however, was the final influence in selecting and fully designing the Methodology, highlighting the vast economic advantages that one SPI strategy may have over another. In fact, Rico's [8] study was the starting point for implementing the Methodology, which quickly picked up a lot of momentum and took on an entire life of its own. After only a few minutes of briefly extending their analyses, the results proved mesmerizingly phenomenal, and thus the Methodology was conceived. In fact, the results of the Methodology, and later the data analyses, exceeded all expectations. And, just to imagine that the final results were preliminarily yielded after only a few moments of additional permutations involving Jones', Kan's and Lazic's study is truly amazing.

2.1 Cost and Benefit Criteria

Three cost criteria and five benefit criteria for a total of eight criteria were chosen with which to evaluate, assess, and analyze SPI alternatives: Training Hours, Training Cost, Effort, Cycle Time, Productivity, Quality, Return-on-Investment, and Break Even Hours. These criteria were chosen because of their commonality and availability as exhibited by Table 1. Reclassification of 487 Metrics for Software Process Improvement (SPI) [2,4,7], Citation Frequency of Metrics for Software Process Improvement(SPI) [4-7], Survey of Software Process Improvement (SPI) Costs and Benefits is given in Table 1 [1,7].

Table 1 Criteria for Evaluating Software Process Improvement (SPI) Alternatives

Criterion	Definition
Training Hours	Training hours refer to the number of person-hours of formal classroom instruction applied for teaching a software process
Training Cost	Training cost refers to the number of training hours plus training fees and travel expenses such as air fare, meals, hotels, car rental, and other applicable training costs
Effort	Effort refers to development effort—the effort required to design, code, unit test, and system test, measured in person-months (Come, Dunsmore, and Shen, 1986)
Cycle Time	Cycle time or duration is defined as the elapsed time in hours or months during which development effort proceeds without interruption (Come, Dunsmore, and Shen, 1986)
Productivity	Productivity is the number of lines of source code produced per programmer-month (person-month) of effort (Come, Dunsmore, and Shen, 1986)
Quality	Quality or defect density is the number of software defects committed per thousand lines of software source code (Come, Dunsmore, and Shen, 1986)
Return-on-Investment	Return-on-investment metrics are collected for the purpose of measuring the magnitude of the benefits relative to the costs (Lim, 1998)
Break Even Hours	Break even hours are defined as the level of activity at which an organization neither earns a profit nor incurs a loss (Garrison and Noreen, 1997)

Survey of Metrics for Software Process Improvement (SPI), showed 74 broad metric classes and 487 individual software metrics. However, Citation Frequency of Metrics for Software Process Improvement (SPI), reclassified the 74 classes of 487 metrics into 11 classes: Productivity (22%), Design (18%), Quality (15%), Effort (14%), Cycle Time (9%), Size (8%), Cost (6%), Change (4%), Customer (2%), Performance (1%), and Reuse (1%).

This helped influence the selection of the eight criteria for SPI cost/benefit analysis, since later quantitative analyses will be based on the existence and abundance of software metrics and measurement data available in published sources. But, availability is not the only reason these eight criteria were chosen. These eight criteria were chosen because it is believed that these are the most meaningful indicators of both Software Process and Software Process Improvement (SPI) performance, especially, Effort, Cycle Time, Productivity, Quality, Return-on-Investment (ROI), and Break Even Hours. Effort simply refers to cost, Cycle Time refers to duration, Productivity refers to number of units produced, Quality refers to number of defects removed, ROI refers to cost saved, and Break Even refers to length of time to achieve ROI [1-4,7,8]. Quality software measurement data will prove to be a central part of this analysis and the direct basis for a Return-on-Investment (ROI) model that will act as the foundation for computing ROI itself. Thus, the Quality criterion is an instrumental factor, and it is fortunate that SPI literature has so abundantly and clearly reported Quality metric and measurement data, despite Quality's controversial and uncommon usage in management and measurement practice [1,7]. The SEI reports that approximately 95.7% of software organizations are below CMM Level 4. CMM Level 4 is where software quality measurement is required. It is safe to assert that 95.7% of software organizations do not use or collect software quality measures.

While, Kan's [2] seminal masterpiece gives a much greater scholarly portrait of sophisticated metrics and models for software quality engineering, Lazic breaks Defect Density down into its most practical terms, Appraisal to Failure Ratio. Lazic has demonstrated that an optimal Appraisal to Failure Ratio of 2:1must be achieved in order to manage software development to the peak of efficiency. While, Kan encourages the use of Rayleigh equations to model defect removal curves, Lazic presents us with the practical saw-tooth form, two parts defects removed before test and one part during test, resulting in very near zero defect levels in finished software products. Since, defects found in test cost 10 times more than defects found before test, and 100 times more after release to customers, Lazic has found that finding 67% of defects before test leads to optimal process performance, minimal process cost, and optimal final software product quality. The other common argument against the use of Defect Density metrics is that they seem to be rather limited in scope, ignoring other more encompassing software life cycle measurements. Again, Lazic's Defect Density Metrics Appraisal to Failure Ratio-based methodology [4-6] has proven that metrics need not be inundating, overwhelming, all encompassing, and sophisticated. This is merely a common confusion between product desirability and Quality. Customer satisfaction and market share measurement is a better form of measuring product desirability while Defect Density is an excellent form of measuring software Quality. Kan [2] gives an excellent exposition of over 35 software metrics for measuring many aspects of software Quality, including customer satisfaction measurement, while reinforcing the strategic nature of Defect Density metrics for measuring software quality associated with SPI.

2.2 Return-on-Investment Model (ROI)

Since very little ROI data is reported [1,8], available, and known for SPI methods, it became necessary to design a new ROI model in order to act as an original source of ROI data, and establish a fundamental framework and methodology for evaluating SPI methods (see Table 2). This original software quality-based ROI model is a direct extension of an earlier work by Rico [8], that was designed for the express purpose of evaluating ROI.

Table 2 Basic Quality-Based Return-on-lnvestment (ROI) Model

	PSP	AT&T Inspection	Basic Inspection	BNR Inspection	GilbHP Inspection	Inspection	Test	Ad Hoc
Software Size	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000
Start Defects	1,000	1,000	1,000	1,000	1,000	1,000	1,000	1,000
Review Efficiency	67%	67%	67%	67%	67%	67%	0%	0%
Review Hours	97	500	708	960	970	1,042	0	0
Review Defects	667	667	667	667	667	667	0	0
Review Detects/Hour	6.86	1.33	0.94	0.69	0.69	0.64	0	0
Review Hours/Defect	0.15	0.75	1.06	1.44	1.46	1.56	0	0
Remaining Defects	333	333	333	333	333	333	1,000	1,000
Test Efficiency	100%	67%	67%	67%	67%	67%	67%	0%
Test Hours	61	1,667	2,361	3,200	3,233	3,472	8,360	0
Test Defects	333	222	222	222	222	222	667	0
Test Defects/Hour	5.47	0.13	0.09	0.07	0.07	0.06	0.08	0
Test Hours/Defect	0.18	7.50	10.63	14.40	14.55	15.63	12.54	0
Validation Defects	1,000	889	889	889	889	889	667	0
Released Defects	0	111	111	111	111	111	333	1,000
Maintenance Hours/Defect	2	75	106	144	146	156	125	125
Development Hours	242	5,088	5,088	5,088	5,088	5,088	5,088	5,088
Validation Hours	158	2,167	3,069	4,160	4,203	4,514	8,360	0
Maintenance Hours	0	8,333	11,806	16,000	16,167	17,361	41,800	125,400
Total Hours	400	15,588	19,963	25,248	25,458	26,963	55,248	130,488
QBreak Even/Ad Hoc	0.78	6.67	6.67	6.67	6.67	6.67	66.67	
PBreak Even/Ad Hoc	6.15	1.65	1.72	1.81	1.81	1.84	10.37	
PBreak Even/Test	14.59	4.79	5.38	6.33	6.38	6.72		
PBreak Even/Inspection	35.96							
Slope (Life Cycle Cost)	25.00	0.64	0.50	0.40	0.39	0.37	0.18	0.08
Y Intercept (w/Investment)	-2000.00	- 12.19	-9.52	-7.53	-7.46	-7.05	- 14.12	
HBreak Even/Ad Hoc	80.25	21.58	22.43	23.56	23.61	23.95	135.27	
HBreak Even/Test	80.58	26.47	29.75	34.99	35.24	37.11		
HBreak Even/Inspection	81.44							
ROI/Ad Hoc	1,290:1	234:1	160:1	114:1	113:1	104:1	10:1	
ROI/Test	430:1	67:1	42:1	27:1	26:1	23:1		
ROI/Inspection	143: 1							

It is a seemingly simple, though intricately complex composite of multiple sub-models, simulating the effects of several SPI methods on efficiency, productivity, quality, cost, break-even points, and ROI. Some of the sub-models represented include a defect removal model and multiple empirical statistical parametric linear and log-linear software cost models. The defect removal model or defect containment analysis model is an experimentally, scientifically, empirically, and commercially validated software quality-based approach to examining SPI method effectiveness and ROI, introduced and used extensively by several major studies [1-4,7,8].

The method involves estimating software defect populations, estimating the efficiency of SPI methods for eliminating software defects, estimating the residual software defect population after applying a particular SPI method, and estimating the cost of eliminating the residual software defect population delivered to customers.

3 Software quality practices and tools

There are many practices and tools that a software vendor can employ [2,7]. There does exist a general framework for understanding how these practices and tools influence the outcomes of software projects, and from which one can determine the potential benefits.

The smart implementation of software quality practices can result in:

• A reduction in software project costs (i.e., an increase in productivity) either directly or through reducing rework.

• The delivery of higher quality software and consequently reducing the customer's cost of ownership

• Reducing time to market (i.e., shorter delivery schedules)

This paper is based on research demonstrating that for specific quality improvement practices and tools, the above benefits can be quantified. Figure 1 shows the chain of effects that follow from the implementation of quality improvement practices and tools. The direct consequence of these practices and tools is increased reuse of software and an improvement in delivered quality.

Increased reuse will lead to higher productivity. Higher quality will lead to lower rework. Increased reuse can also influence quality. There is some earlier evidence showing that quality improves by reusing pre-existing artifacts [7]. More recently, one study found that object-oriented classes tend to have a lower defect density than classes that were new or reused with modification [2]. In general, it was found that reuse reduced the amount of rework. Of course, if rework is reduced, then productivity increases as well.

An improvement in delivered software quality reduces the total cost of ownership of the software by the customer. Therefore, quality practices have a direct impact on the ongoing costs that are incurred by the customer while operating the software.

It is not atypical that 50% or more of a project's cost can be rework [5]. Rework means fixing defects. If the quality of the software is higher then less effort will be spent on rework since fewer defects need to be fixed. Higher development productivity and lower rework result in reduced overall software project costs. And, lower total costs mean less effort by the project staff. Higher productivity also translates into a reduction in overall project schedule.

In the remainder of this paper we will explain how these mechanisms operate and provide examples.

3.1 Quality and The Cost of Ownership

In this section we will present a model for calculating the relationship between software quality and the cost of ownership of software. Through a number of examples, it will be demonstrated that low quality software can be quite expensive for the customer.

A Definition of Customer Costs

When software is purchased, the customer costs consist of three elements:

1. Pre-purchase costs constitute the resources expended by the customer to investigate different software companies and products.
2. Installation costs constitute the resources invested in installing and verifying the operation of new software beyond that which is covered in the purchase contract (i.e., as part of the purchase price).
3. Post-purchase costs constitute the resources to deal with software failures, including consultants and the maintenance of redundant or parallel systems.

For our purposes we will focus mainly on the latter two cost categories as being the most relevant. Both cost categories increase when there are defects in the software.

Installation defects typically are due to interface problems with legacy systems or interoperability with pre-installed software. For commercial (shrink-

wrapped) software, installation problems are usually minimal. However, if there are problems during installation, the customer bears the cost of trying to figure out the problem and making support calls to the developers. For customized software there are greater chances of installation difficulties. Dealing with them may entail expenditures on consultants and third party integrators. Additional cost components here are those of lost productivity due to the software not being installed on time, and lost sales or revenue due to the unavailability of an operational system.

Post-purchase costs occur because the customer has to deal with failures of the software or lower performance than expected. This entails interacting with the vendor to describe and recreate the problem, and implementing workaround procedures (which may be inefficient) while waiting for the fix. Bugs may result in the loss of data and the need to re-enter them, and may result in capital costs being incurred due to early retirement of new but ineffective systems. Another cost of failures is that of operating redundant or parallel systems at the same time as the new software until the new software is fully functional with no quality problems.

These costs tend to be different depending on whether the customer company is small or large. Small companies tend to incur relatively larger costs due to software defects because [3]:

• Smaller companies are less likely to have internal support staff who can help troubleshoot and correct errors. This means that the customer has to go back to the vendor, which takes longer. The consequence is that the disruptive effects of defects last longer.
• Larger corporations tend to get higher priority from the software vendors.

Therefore their requests for fixes, patches, and workarounds get more immediate attention than smaller companies. Also, large companies are more likely to have internal support staff. Therefore the disruptions from discovering problems are minimized.

For instance, in the manufacturing sector in USA it has been estimated that the software defect costs per employee varies from US$1466.1 to US$2141.4 for small companies (less than 500 employees), and from US$128.9 to $277.8 for large companies (more than 500 employees) [3]. For a 100 person company that translates to US$146,614 to US$214,138, and for a 10,000 employee company that makes US$1,289,167 to US$2,777,868 per year.

3.2 Savings From Higher Quality Software

We can derive a model to estimate the savings to the customer from purchasing higher quality software. The scenario is that of a customer choosing between two products from two vendors. The products are to perform the same functions.
To develop a model that is actionable and where sufficient benchmark data is available that can be used, certain assumptions need to be made. The approach that we take is to make assumptions that are conservative. That means we will make assumptions which result in the estimate of savings from better quality being underestimated rather than being exaggerated. This way we develop a lower bound model. The installation and post-purchase costs for a customer who has bought a software product are given by:

$$Customer\ Cost = Q \times S \times C \times P \quad (1)$$

Where:
Q The quality of the product defined in terms of defect density. Defect density is total defects found after release per one thousand lines of code.
S This is the size of the product in thousands of lines of code or Function Points (or some other size measure).
C The cost per defect for the customer. This is due to installation and post-purchase problems.
P The proportion of defects that the customer finds. It is defined as the fraction of all of the defects in the software that a single customer is likely to find. For example, if it is 0.01, it means that any single customer is likely to experience 1% of all of the detectable defects in the software.

Let us assume that we have two software products, **A** and **B**, that have the same functionality (i.e., say the Function Points count is the same). Let us also say that **A** has higher quality than **B** (i.e., it has a lower defect density). These two products are of the same size and will be used the same way. A customer can choose between these two products. The percentage cost savings to the customer from using product **A** as opposed to product **B** would be:

$$Percentage\ Saving = \frac{[Customer\ Cost_B] - [Customer\ Cost_A]}{[Customer\ Cost_B]} \times 100 \quad (2)$$

For example, if the percentage saving is 20%, then this means that the specific cost to the customer of owning product **A** (the installation and post-purchase costs) is 20% less than that of owning product **B**. We can easily make the assumption that the sizes for **A** and **B** are the same since we are talking about the same system functionality.

Therefore, the percentage savings equation becomes:

$$Percentage\,Saving = \frac{[Q_B \times C_B \times P_B] - [Q_A \times C_A \times P_A]}{[Q_B \times C_B \times P_B]} \times 100$$

We can simplify this to:

$$Percentage\,Saving = \left(1 - \frac{[Q_A \times C_A \times P_A]}{[Q_B \times C_B \times P_B]}\right) \times 100$$

Let:

P_A The proportion of defects that the customer finds in program **A**.

P_B The proportion of defects that the customer finds in program **B.**

And, since **B** has more defects than **A**, we would also expect to see:

$$P_B > P_A$$

Then, it is clear that:

$$\left(1 - \frac{Q_A \times C_A}{Q_B \times C_B}\right) < \left(1 - \frac{Q_A \times P_A \times C_A}{Q_B \times P_B \times C_B}\right)$$

Therefore, if we use the following equation, we are actually calculating a lower bound on the percentage saving:

$$Percentage\,Saving = \left(1 - \frac{Q_A \times C_A}{Q_B \times C_B}\right) \times 100$$

The component of the above equation (cost per defect) has two elements: mitigation costs which are the costs that the customers incur in response to defects actually manifesting themselves, and avoidance costs such as installation costs and redundant system costs. There is evidence that the mitigation costs of the customers are linearly proportional to the reduction in defects [3]. This means that as the number of defects detected decreases, the customer costs decrease proportionally. Therefore, for these mitigation costs, we can treat as a constant (i.e.,).However, the avoidance costs decrease non-linearly with defects [3]. In fact, in many instances the benefits of reduced defects do not really accrue until the defect counts approach zero. For example, redundant systems have to be kept in place even if one defect remains because one defect is sufficient to bring down the whole system, which would then require the redundant system to kick in. Similarly, some of the customer costs will not disappear even if the software did have zero defects. For instance, there will always be installation costs even for perfect software because some of the legacy applications that the new software integrates with may have defects. Therefore, for these avoidance costs we make the conservative assumption that a reduction in defects will have no impact on the customer since in most cases the true defect density will be unlikely to approach zero. In terms of our model, this means that the component in the above equation consists only of the mitigation costs. Based on the recent data collected in [3], we can make another conservative assumption that 75% of the post-purchase customer costs are mitigation costs, and the remaining 25% are avoidance costs. This means that if the defect content of the software went all the way to zero, 25% of the current post-purchase costs would still be incurred. To demonstrate the conservatism in this assumption, the NIST report [3] notes that the cost per defect in the manufacturing sector is on averageUS$4,018,588. The total of installation, acceptance, maintenance, and redundant system costs per firm is US$258,213.6. If, on average a firm experiencing defects has 40 major ones per year, then the mitigation costs can actually be quite large compared to avoidance costs according to these numbers. Therefore, if we say that 75% of the customer costs are mitigation costs that decrease proportionally to defect reduction, then the above equation becomes:

$$Percentage\,Saving = \frac{Q_B - Q_A}{Q_B} \times 100 \times 0.75 = \left(1 - \frac{Q_A}{Q_B}\right) \times 75 \quad (3)$$

Which gives a lower bound on the post-purchase cost savings from improved quality.

There are two interesting things to note about Eqn. (3). First, in practice it is quite easy to get comparative values for **Q_A** and **Q_B**. Even if **Q_A** and **Q_B** are not readily available, there are reliable ways to estimate their ratio using black-box testing techniques. Therefore, it provides us with a very actionable model. Second, the excessive conservatism that we have emphasized in deriving our assumptions, as you will see, do not dilute the strong conclusions that can be drawn about the cost of quality to the customer. We now look at a number of examples to illustrate the benefit of better quality software from the customer perspective using benchmark data.

3.3 Benchmarking Customer Costs

The following examples are based on published data and illustrate the savings to the customer under a diverse set of scenarios. The savings are calculated according to Eqn. (3).

SW-CMM Example

Jones [7] has published the defect density of software from companies at different maturity levels as measured on the Capability Maturity Model for Software. If we consider the average values, we can

construct Table 3, which shows the reduction in the customer's cost as the maturity of the supplier increases. For instance, software from companies at ML3 is 27.75% cheaper from the customer's perspective compared to software from an ML1 company. We can also see that there are dramatic reductions in the customer cost as the maturity of the supplier reaches the higher levels.

Table 3 Percentage reduction in ownership costs due to improved quality from the customer's perspective

	ML1	ML2	ML3	ML4	ML5
ML1		12.75%	27.75%	52.5%	64.5%
ML2			18%	47.25%	62.25%
ML3				39%	58.5
ML4					40.5%
ML5					

Customer Savings By Industry, Country, and Platform

We can also compare the customer cost when buying software from average companies in their domain versus best-in-class ones. Jones [7] identifies delivered defects per Function Point for average and best-in-class companies in a number of different business domains. Best-in-class are the top performers in their domain (defined as the top 10%). Using those numbers we can determine the cost savings to a customer from buying software from an average company compared to a best-in-class company. These results are shown in Table 4. As is obvious, there are quite dramatic benefits to the customer from buying software from a best-in-class company that delivers high quality software.

Table 4 The percentage customer cost reduction between the average and best-in-class companies in each business sector at the three project sizes.

	Small Projects	Medium Projects	Large Projects
MIS	62.25%	66.75%	55.5%
Systems Software	71.25%	61.5%	59.25%
Commercial	71.25%	63%	55.5%
Military	71.25%	69%	58.5%

Data are from MIS and commercial projects, a small project is 100 FP, a medium project is 1000 FP, and a large project is 10000FP. For systems software and military projects, a small project is1000FP, a medium project is 10000FP, and a large project is 100000FP. This data is derived from [7].

The savings in ownership costs can be quite large compared to average producers (assuming a bell curve, one can make the reasonable assumption that at least half of the vendors in each category are at or below average). Based on defect density data collected during the most recent ISBSG benchmark we can evaluate the difference to the customer in acquiring software from best-in-class companies (in the top 10% as measured by delivered defect density) and average performers. We also compare best-in-class to worst performers (bottom 10% as measured by delivered defect density).

Table 5 Cost savings to the customer in terms of buying software from the best performers vs. average and worst performers within each of these countries.

	Best-in-Class vs. Average Performers	Best-in-Class vs. Worst Performers
Australia	53.25%	69.75%
Canada	66%	69.75%
India	72%	74.25%
Japan	63.75%	74.25%
Netherlands	60%	69%
United Kingdom	60.75%	74.25%
United States	52.5%	70.5%

Data from table means that, for example, if an Australian customer buys software from a company that delivers best-in-class projects, then their post-purchase costs would be about 70% cheaper compared to if they buy it from a worst performer and over 50% better compared to if they buy it from an average performer.

Table 6 Cost savings to the customer in terms of buying soft-ware from the best performers vs. average and worst performers within each of these business areas.

	Best-in-Class vs. Average Performers	Best-in-Class vs. Worst Performers
Sales	51.75%	63.75%
Financial (excluding banking)	65.25%	74.25%
Insurance	49.5%	68.25%
Accounting	51.75%	63%
Inventory	34.5%	73.5%
Banking	63.75%	70.5%
Manufacturing	61.5%	74.25%
Telecommunications	69%	73.5%
Marketing	64.5%	74.25%
Legal	54%	72%
Engineering	65.25%	74.25%

This table shows the cost savings to the customer in terms of buying software from the best performers vs. average and worst performers within each of these target platforms. This means that, for example, if a mainframe customer buys software from a company that delivers best-in-class projects, then their post-purchase costs would be over 70% cheaper compared to if they buy it from a worst performer and around 64% better compared to if they buy it from an average performer.

4 Economics Of Software Quality

Cost of quality represents any and all costs that organization incurs from having to repeat a process more than once in order to complete the work correctly. Cost of developing software Quality (CoSQ) is useful to enable our understanding of the economic trade-offs involved in delivering good-quality software. Commonly used in manufacturing, its adaptation to software offers the promise of preventing poor quality but, unfortunately, has seen little use to date. Different authors and researcher have used different ways to classify components for quality cost [2-8], if we look carefully their understanding about various components are approximately the same as shown in Fig.2.

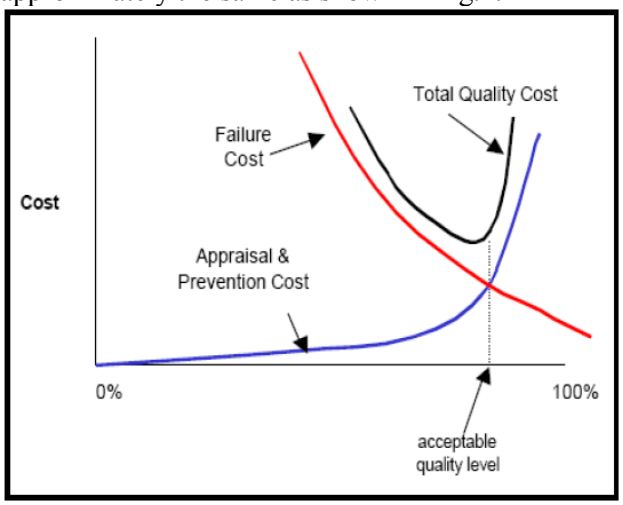

Fig. 2 Model of Cost of software Quality (CoSQ)

4.1 Statement Of the Problem

A key metric for measuring and benchmarking the software testing efficacy is by measuring the percentage of possible defects removed from the product at any point in time. Both a project and process metric – can measure effectiveness of quality activities of a all over project by:

$$DRE = E/(E+D) \quad (4)$$

Where E is the number of errors found before delivery to the end user, and D is the number of errors found after delivery. The goal is to have DRE close to 100%. The same approach is applied to every test phase denoted with ***i*** as shown on Fig. 3:

$$DRE_i = E_i / (E_i + E_{i+1}) \quad (5)$$

Where E_i is the number of errors found in a software engineering activity ***i***, and E_{i+1} is the number of errors that were traceable to errors that were not discovered in software engineering activity ***i***.

The goal is to have this DRE_i approach to 100% as well i.e., errors are filtered out before they reach the next activity. Projects that use the same team and the same development processes can reasonably expect that the *DRE* from one project to the next are similar. For example, if on the previous project, you removed 80% of the possible requirements defects using inspections, then you can expect to remove ~80% on the next project. Or if you know that your historical data shows that you typically remove 90% before shipment, and for this project, you've used the same process, met the same kind of release criteria, and have found 400 defects so far, then there probably are ~50 defects that you will find after you release. How to combine Defect Detection Technique (DDT) to achieve high *DRE*, let say >85%, as a threshold for SPI required effectiveness [4-6] which describe optimum combination of software defect detection techniques choices.

Note that the defects discussed in this section include all severity levels, ranging from severity 1: activity stoppers, down to severity 4. Obviously, it is important to measure defect severity levels as well as recording numbers of defects.

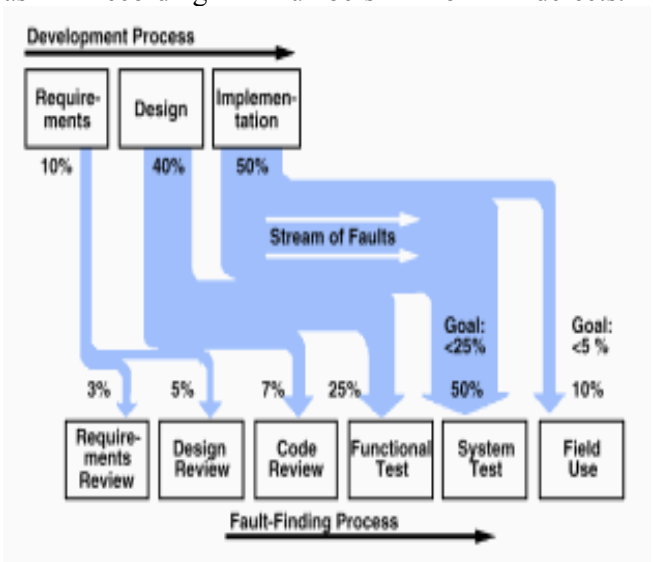

Fig. 3 Fault Injection and Fixing Model

4.2 The Real Cost Of Software Defects

It is obvious that the longer a defective application evolves the more costly it is to repair. But how much more? The answer might surprise you. According to the collected metrics of one software development organization, a bug that costs $1 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program, and many thousands of dollars if it is identified after the software has been deployed in the field [7], as described on Fig. 4. Several studies [2-6] has published over nearly three decades that demonstrate how the cost for removing a software defect grows exponentially for each downstream phase of the development lifecycle in which it remains undiscovered.

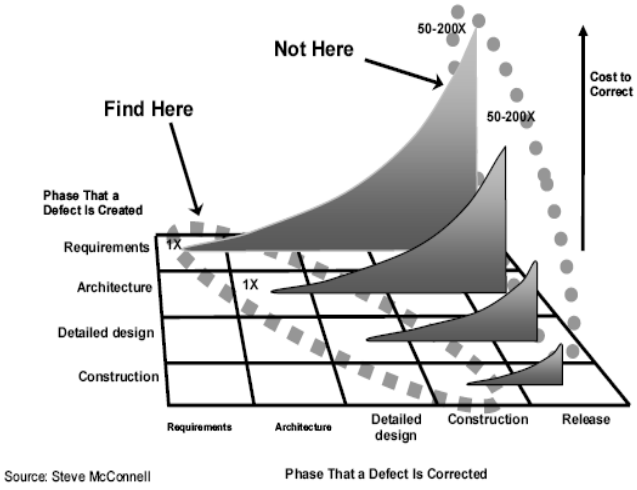

Fig. 4 Engineering Rules for Cost Of Defect Removal [3]

Further, another major research project conducted recently by the United States Department of Commerce, National Institute of Standards and Technology (NIST) [3] showed that in a typical software development project, fully 80% of software development dollars are spent correcting software defects. The same NIST study also estimated that software defects cost the U.S. economy, alone, $60 billion per year. Many organizations view the software development lifecycle, in a Conventional way, as a linear process with discrete functions: design, develop, test and deploy. In reality, the software development lifecycle is a cyclical function with interdependent phases. Quality assurance has a role in every phase of that lifecycle, from requirements review and test planning, to code development and functional testing, to performance testing and on into production. It was unanimously agreed that quality and quality assurance is more than strictly testing at the end of the development process. Starting quality initiatives early and paying attention to quality throughout the development, deployment and production effort is key in order to achieve a baseline goal of zero-defect software.

4.3 Software Testing Economics

4.3.1 Techniques To Analyze Return On The Testing Investment (ROTI)

The ROTI model compares the development cost for a conventional project with the development cost for a project that uses Test Driven Development (TDD) as depicted on Fig. 5. The investment cost is the additional effort necessary to complete the TDD project as compared to the conventional project. The life cycle benefit is captured by the difference in quality measured by the number of defects that the TDD team finds and fixes, but the conventional project does not. This defect difference is transformed into a monetary value using the additional developer effort corresponding to finding and fixing these defects in the conventional project. The concepts of the life cycle benefit and the investment cost in our context are depicted in Fig. 10. The upper horizontal line corresponds to the conventional project with additional quality assurance phase! The lower horizontal line corresponds to the TDD project. Our model captures the return on investment for an experienced TDD team in software testing process improvement (SPI).

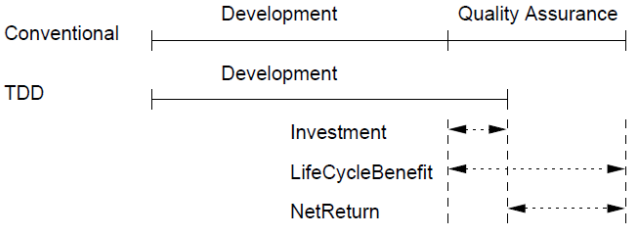

Fig. 5 Overview of benefit cost ratio calculation

4.3.2 Financial ROI

From a developer's perspective, there are two types of benefits that can accrue from the implementation of good software quality practices and tools: money and time. A financial ROTI looks at cost savings and the schedule ROTI than looks at schedule savings. Direct financial ROTI is expressed in terms of effort since this is the largest cost on a software project. There are a number of different models that can be used to evaluate financial ROTI for software quality.

The first is the most common ROI model. We will show that this model is not appropriate because it does not accurately account for the benefits of investments in software projects. This does not

mean that that model is not useful (for instance, accountants that we speak with do prefer the traditional model of ROI), only that we will not emphasize it in our calculations.

Methods for return on investment (ROI) include benefit, cost, benefit/cost ratio, ROI, net present value, and breakeven point are given in Fig. 6. ROI methods in general are quite easy, indispensable, powerfully simplistic, and absolutely necessary in the field of software process improvement (SPI). It is ironic that ROI methods are not in common practice. The literature does not abound with ROI methods for SPI. The ROI literature that does exist is very hard to locate, appears infrequently, and is often confusing.

Metric	Definition	Formula
Costs	Total amount of money spent on a new and improved software process	$\sum_{i=1}^{n} Cost_i$
Benefits	Total amount of money gained from a new and improved software process	$\sum_{i=1}^{n} Benefit_i$
B/CR	Ratio of benefits to costs	$\frac{Benefits}{Costs}$
ROI	Ratio of adjusted benefits to costs	$\frac{Benefits - Costs}{Costs} \times 100\%$
NPV	Discounted cash flows	$\sum_{i=1}^{Years} \frac{Benefits_i}{(1+Discount\ Rate)^{Years}} - Costs_0$
BEP	Point when benefits meet or exceed cost	$\frac{Costs}{Old\ Costs / New\ Costs - 1}$

Fig. 6 ROI metrics showing simplicity of ROI formulas and their order of application

We also look at ROI at the project level, specially on return on the testing investment (ROTI), rather than at the enterprise level. ROI at the enterprise level (or across multiple projects) requires a slightly different approach which we will not address directly here.

The most common ROI model, and that has been used more often than not in software engineering, we use for ROTI calculation is:

$$ROTI_1 = \frac{Total \cdot CoQ \cdot Saved - Test \cdot Investment}{Test \cdot Investment} \quad (6)$$

This ROTI model gives how much the Total Cost of Quality (CoQ) savings gained from the project were compared to the initial investment. Let us look at a couple of examples to show how this model works. We will use a hypothetical case study to illustrate the use of this cost of quality technique to analyze return on the testing investment. Suppose we have a software product in the field, with one new release every quarter. On average, each release contains 1,000 "must-fix" bugs—unacceptable defects—which we identify and repair over the life of the release. Currently, developers find and fix 250 of those bugs during development, while the customers find the rest. Suppose that you have analyzed the costs of internal and external failure. Bugs found by programmers costs $10 to fix. Bugs found by customers cost $1,000 to fix. We analyze three cases of software development and testing process which provide Low Quality, Good Quality and High Quality Results.

Case 1: Low Quality Results

Case 1 is assumed to be a fairly small systems software project of 251 function points in size. Defect potentials are derived by raising the function point total of the application to the 1.25 power, which results in a total of 1,000 defects or 4 defects per function point [2]. Defect removal efficiency is assumed to be 75% overall. The development team is assumed to be below level 1 on the CMM scale in Software Development Process (SDP) which is unpredictable and poorly controlled i.e. Ad hoc level.

As shown in the "Case 1 Testing" column in Fig. 7, our cost of quality is three-quarters of a million dollars. It's not like this $750,000 expenditure is buying us anything, either. Given that 750 bugs escape to the field, it's a safe bet that customers are mad!

Case 2: Good Quality Results

Case 2 is exactly the same size and the same class of software as Case 1. The project management decided to improve software testing process (STP) and invested in testing staff $60,000 and test infrastructure $10,000 as shown in the "Case 2 Testing" column in Fig. 7.

The development team is assumed to be level 1 on the CMM scale. Defect removal efficiency is assumed to be 85% overall. Defect removal operations consist of six test stages: 1) unit test, 2) new function test, 3) regression test, 4) integration test, 5) system test, and 6) external Beta test.

Case 3: High Quality Results

Case 3 is exactly the same size and the same class of software as Case 1. The development team is assumed to be higher than level 3 on the CMM scale. By means of more effective defect prevention such as Quality Function Deployment (QFD) and

Six-Sigma the defect potentials are lower. Defect removal efficiency is assumed to be 95%. Defect removal operations consist of nine stages: 1) design inspections; 2) code inspections; 3) unit test, 4) new function test, 5) regression test, 6) integration test, 7) performance test, 8) system test, 9) external Beta test.

To clarify the differences between the three case studies, note that both examples are exactly the same size, but differ in these key elements:

- CMM levels
- Defect prevention
- Defect potentials
- Defect removal efficiency
- Development schedules
- Development effort
- Development costs

	A	B	C	D
1	**Testing Investment Options: ROI Analysis**			
2				
3				
4	Testing resources	Case 1	Case 2	Case 3
5		CMM <1 Level	CMM 1 Level	CMM >3 Level
6	Staff	$0	$60,000	$60,000
7	Infrastructure	$0	$10,000	$10,000
8	Tools	$0	$0	$12,500
9	**Total Test Investment**	**$0**	**$70,000**	**$82,500**
10				
11	Development (Requirement, Design, Code)			
12	Must-Fix Bugs Found	250	250	350
13	Fix Cost - $10 per bug (Internal Failure)	$2,500	$2,500	$3,500
14				
15	Testing			
16	Must-Fix Bugs Found	0	600	600
17	Fix Cost - $100 per bug (Internal Failure)	$0	$60,000	$60,000
18				
19	Customer Support			
20	Must-Fix Bugs Reported	750	150	50
21	Fix Cost - $1000 per bug (External Failure)	$750,000	$150,000	$50,000
22				
23				
24	Cost of Quality (CoQ)			
25	Conformance	$0	$70,000	$82,500
26	Nonconformance	$752,500	$212,500	$113,500
27	Total CoQ	$752,500	$282,500	$196,000
28				
29	Return on Investment (ROI_1)	#N/A	571%	575%
30				
31	Return on Investment (ROI_2)	#N/A	62%	74%

Fig. 7 Using Cost of Quality to Analyze two ways of Return on Investment calculation

Suppose we calculate that bugs found by testers would cost $100 to fix. This is one-tenth what a bug costs if it escapes to our customers. So, we invest $70,000 per quarterly release in a Case 2 testing process. The "Case 2 Testing" column shows how profitable this investment is. The testers find 600 bugs before the release, which cuts almost in 80% the number of bugs found by customers. This certainly will make the customers happier. This process improvement will also make the Chief Financial Officer happier, too: Our total cost of quality has dropped to about half a million dollars and we enjoy a nice fat 571% return on our $70,000 investment.

In some cases, we can do even better. For example, suppose that we invest $12,500 in test automation tools and Inspection activities (see in the "Case 3" column in Fig. 7). Let's assume we intend to recapture a return on that investment across the next twelve quarterly releases. Would we be happy if that investment in test automation helped us find about 67% more bugs?

Finding 350 bugs in development phases and 600 bugs in the test process would lower the overall customer bug find count for each release to 50. Deployment of more formal and rigorous STP in which 950 bugs out of 1000 were removed, i.e. Total DRE 95%. Certainly, customers would be much happier to have the more-thoroughly tested system. In addition, cost of quality would fall to a little under $200,000, a 575% return on investment (ROI).

4.4 The Life Cycle Benefit model parameters formulas for calculations

This section describes those formulas of our *OptimalSQM* metrics model [6] which are necessary to understand the break-even and ROTI analysis if the investment of described STP improvements in previous sections pays off.

Calculating the return on investment ROI means to add up all the benefits of the investment, subtract the cost, and then compute the ratio of the cost according the equation (6) in Section 4.3.2 Financial ROI. If the investment in STP improvement pays off, the $ROTI_1$ is positive, otherwise negative. In our evaluation of TDD we focus on the benefit cost ratio BCR which is easily derived from the return on investment.

$$BCR = LifeCycleBenefit/Investment = ROTI_1 + 1$$

Studying the BCR instead of the $ROTI_1$ makes the break-even analysis much simpler, see below.

4.4.1 Investment Cost

We first look at the investment cost. For the conventional project, the development phase includes design, implementation and test. The

development phase of the TDD project is comprised only of test-driven development.
As first empirical evidence suggests, we assume that the TDD project lasts longer than the conventional project. We call the ratio of the project durations the test-speed-disadvantage (TSD).

$$TSD = Time_{Conv}/Time_{TDD}$$

Since we assume that the development phase is shorter for the conventional project, because include small number of test activities, the test-speed-disadvantage ranges between 0 and 1:0 < TSD < 1.

Using productivity figures to explain the difference in elapsed development time between the two kinds of project, the TDD development is (1 − TSD) × 100 % less productive than the conventional project. Finally, the investment is the difference between the development cost of the TDD project and the conventional project as depicted in Fig. 5.

4.4.2 Life Cycle Benefit

Now, we consider the benefit. Each development process is characterized by a distinct defect-removal-efficiency - DRE (recall the section 4.1). The defect-removal-efficiency denotes the percentage of defects a developer eliminates during development. Initially, a developer inserts a fixed amount of defects per thousands lines of code (initial-defect-density, IDD), but he eliminates DRE × 100 % of the defects during the development process. From the increased reliability assumed for TDD,we have:

$$0 < DRE_{Conv} < DRE_{TDD} < 1.$$

The additional quality assurance (QA) phase of the conventional project compensates for the reduced defect-removal-efficiency of the conventional process. The only purpose of the Comprehensive QA plan phase is to remove all those defects found by TDD but not by the conventional process (recall the *Case 3* in section 4.3.2**)**. The amount of defects to be removed in the Comprehensive QA plan phase is mainly characterized by:

$$\Delta DRE = DRE_{TDD} - DRE_{Conv}.$$

The benefit of TDD is equal to the cost of the Comprehensive QA plan phase for the conventional project. The benefit depends on the effort (measured in developer months) for repairing one line of code during QA, which is characterized by

$$\mathrm{QA}_{\mathrm{Effort}} = \frac{DRT * IDD}{WT}$$

QA_{Effort} depends on the following:
• The defect removal time DRT. It describes the developer effort in hours for detecting (finding) and removing one defect.
• The initial defect density IDD. The number of de-facts per line of code inserted during development.
• The working time WT. The working hours per month of a developer. The reciprocal of QA_{Effort} is a measure for the productivity during the QA phase.

4.4.3 Benefit Cost Ratio

The benefit cost ratio is the ratio of the benefit and the investment. Substituting the detailed formulas given in [6] of our model, the benefit cost ratio becomes:

$$\mathrm{BCR} = \frac{QA_{Effort} * \mathrm{Prod} * \Delta DRE * TSD}{(1 - TSD)} \qquad (7)$$

Where, **Prod** is the productivity of the conventional project during the development phase measured in lines of code per month. Values larger than 1 for the BCR mean a monetary gain from TDD, values smaller than 1 a loss.

4.4.4 Break Even

Setting the benefit cost ratio equal to 1, we get a relation between the test-speed-disadvantage of TDD and the reliability gain of TDD:

$$\mathrm{TSD} = \frac{1}{c * \Delta DRE + 1}, \text{ or}$$

$$\Delta DRE = \frac{1 - TSD}{c * TSD}, \text{ where } c = QA_{Effort} * Prod$$

As an example, we examine the benefit cost ratio of the following scenario.

Factor	Value
DRT	10 h/defect
IDD	0.1 defects/LOC
WT	135 h/month
Prod	350 LOC/month

Let TSD and ΔDRE vary. Figure 8 shows the benefit cost ratio plane spanned by the test-speed-disadvantage TSD and the defect-removal-efficiency difference ΔDRE. Values larger than 4 are cut off.
For large values of the test-speed-disadvantage (TSD > 0.9) the TDD project performs almost

always better than the conventional project, even for a small defect-removal-efficiency difference. On the other hand, if the test-speed-disadvantage is very small (TSD < 0.2), TDD does not produce any benefit regardless how large the defect-removal-efficiency difference is.

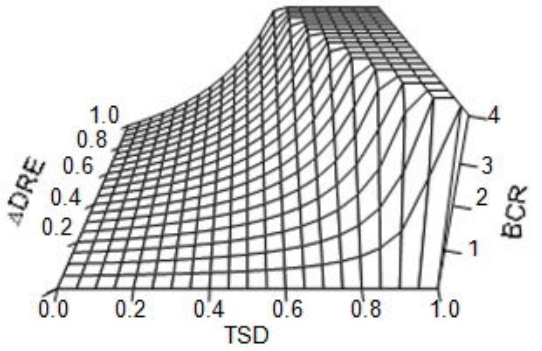

Fig. 8 Benefit cost ratio dependent on TSD and ΔDRE

The TSD can be estimated with formula (10) for ΔSCED in next Section. The relationship between cost savings defined by $ROTI_2$ and schedule reduction is shown in Fig. 10 in the same Section.

4.4.5 Schedule Benefits

If software quality actions are taken to reduce development cost, then this will also lead to a reduction in development schedule. We can easily calculate the reductions in the development schedule as a consequence of reductions in overall effort. In this section we will outline the schedule benefits of quality improvements. To do so we will use the schedule estimation model from COCOMO [2].

It is instructive to understand the relationship between project size and schedule as expressed in the COCOMO II model [2]. This is illustrated in Fig. 9. Here we see economies of scale for project schedule. This means that as the project size increases, the schedule does not increase as fast. The three lines indicate the schedule for projects employing different levels of practices. The lower risk and good practice projects tend to have a lower schedule.

Another way to formulate the ROTI model in Eqn. 6 which will prove to be handy is:

$$ROTI_2 = \frac{Original \cdot Total \cdot CoQ - New \cdot Total \cdot CoQ}{Original \cdot Total \cdot CoQ} \quad (8)$$

The New Total CoQ is defined as the total cost of software quality the project delivered after implementing the quality improvement practices or tools as in our Case 2 and Case 3. This includes the cost of the investment itself. Let us look at some examples. For Case 2 we have:

$$ROTI_2 = \frac{\$752{,}500 - \$282{,}500}{\$752{,}500} = 0.62 = 62\%$$

This means that in Case 2 project, the investment only saved 62% of overall project cost.

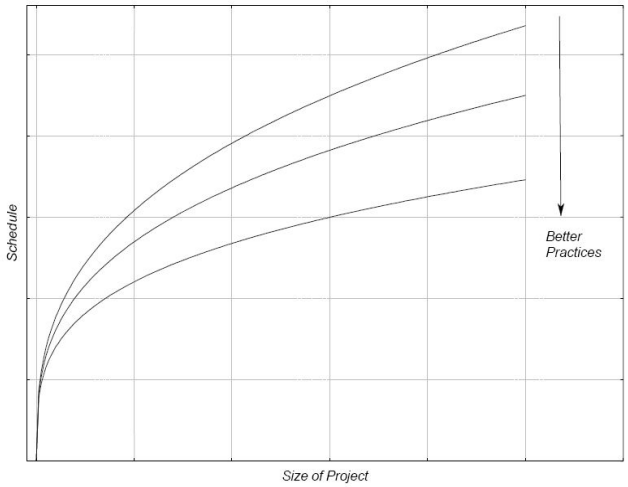

Fig. 9 Relationship between project size and schedule in COCOMO II.

Now for Case 3 we have:

$ROTI_2 = \frac{\$752{,}500 - \$196{,}000}{\$752{,}500} = 0.74 = 74\%$, i.e. the same investment saved 74% of overall project cost.

We can then formulate the New Total CoQ as follows:

$$New \cdot Total \cdot CoQ = Original \cdot Total \cdot CoQ \cdot (1 - ROTI_2)$$

Now, we can formulate the schedule reduction (ΔSCED or SCEDRED) as a fraction (or percentage) of the original schedule as follows:

$$\Delta SCED = \frac{Original \cdot Schedule - New \cdot Schedule}{Original \cdot Schedule} \quad (9)$$

By substituting the COCOMO equation for schedule, we now have:

$$\Delta SCED = \frac{PM_{Original}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)} - PM_{New}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)}}{PM_{Original}^{0.28+(0.002\times\sum_{j=1}^{5} SF_j)}}$$

where:

$PM_{Original}$ - The original effort for the project in person-months.
PM_{New} - The new effort for the project (after implementing quality practices) in person-months

SF_j - A series of five Scale Factors that are used to adjust the schedule (precedentedness, development flexibility, architecture / risk resolution, team cohesion, and process maturity).

Now, by making appropriate substitutions, we have simplified Eqn.:

$$\Delta SCED = 1 - (1 - ROTI_2)^{0.28 + (0.002 \times \sum_{j=1}^{5} SF_j)} \quad (10)$$

The relationship between cost savings and schedule reduction is shown in Fig. 10. As can be seen, the schedule benefits tend to be at smaller proportions than the cost benefits. Nevertheless, shaving off 10% or even 5% of your schedule can have nontrivial consequences on customer relationships and market positioning.

4.4.6 Interpreting The ROI Values

In this section we will explain how to interpret and use the ROI values that are calculated. First, it must be recognized that the ROI calculations, cost savings, and project costs as presented in our models are estimates. Inevitably, there is some uncertainty in these estimates. The uncertainty stems from the variables that are not accounted for in the models (there are many other factors that influence project costs, but it is not possible to account for all of these since the model would then be unusable). Another source of uncertainty is the input values themselves.

These values are typically averages calculated from historical data; to the extent that the future differs from the past these values will have some error. Second, note that the calculated ROI values are for a single project. A software organization will have multiple on-going and new projects. The total benefit of implementing software quality practices to the organization can be calculated by generalizing the results to the organization. For example, if the ROI for a single project is say a 15% saving. Assuming that the input values are the same for other projects in the organization, then we can generalize to the whole organization and estimate that if software quality practices are implemented on all projects in the organization, the overall savings would be 15%.

If the software budget for all the projects is say 20 million, then that would translate into an estimated saving of 3 million. Note that this is not an annual saving, but a saving in total project budgets hat may span multiple years (i.e., for the duration of the projects). To annualize it then the 15% savings must be allocated across multiple years. If you are implementing quality improvement on a single project, then these costs would have to be deducted from the single project savings. If you are implementing quality practices in the whole organization, then these costs will be spread across multiple projects. In such a case, these costs would be deducted from the organizational savings (the calculation of which is described above).

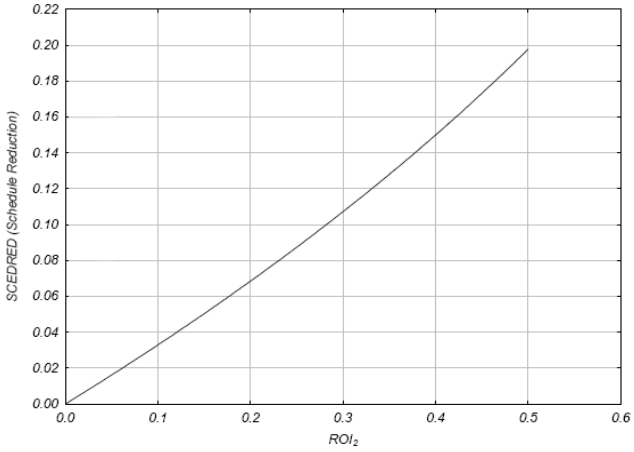

Fig. 10 The relationship between cost savings and schedule reduction for up to 50% cost savings. The assumption made for plotting this graph was that all Scale Factors were at their nominal values.

5 Conclusion

The data in this paper, especially taking into account the conservatism in the assumptions made, provides compelling evidence that substantial reductions in post-purchase costs can be gained by focusing on higher quality suppliers. In fact, one would contend that in many cases the savings would exceed the actual initial and annual licensing costs of the software. An improvement in delivered software quality reduces the total cost of ownership of the software by the customer. Therefore, quality practices have a direct impact on the ongoing costs that are incurred by the customer while operating the software.

It is not atypical that 50% or more of a project's cost can be rework [5]. Rework means fixing defects. If the quality of the software is higher then less effort will be spent on rework since fewer defects need to be fixed. Higher development productivity and lower rework result in reduced overall software project costs. And, lower total costs mean less effort by the project staff. Higher

productivity also translates into a reduction in overall project schedule.

Small companies tend to incur relatively larger costs due to software defects because [3]:

• Smaller companies are less likely to have internal support staff who can help troubleshoot and correct errors. This means that the customer has to go back to the vendor, which takes longer. The consequence is that the disruptive effects of defects last longer.

• Larger corporations tend to get higher priority from the software vendors.

Therefore their requests for fixes, patches, and workarounds get more immediate attention than smaller companies. Also, large companies are more likely to have internal support staff. Therefore the disruptions from discovering problems are minimized.

There is accumulating evidence that higher maturity organizations produce higher quality software [1]. Therefore, it would pay to seek higher maturity suppliers and demand that existing uppliers invest in improving their maturity levels.

This paper satisfies these objectives by designing, constructing, and exercising a multi-part methodology consisting of a Defect Removal Model, Cost and Benefit Data, Return-on-Investment Model, Break Even Point Model, and Costs and Benefits of Alternatives, which all lead up to a Cost and Benefit Model.

References

[1] El-Emam, K. and D. Goldenson (2000). "An Empirical Review of Software Process Assessments." Advances in Computers, 53: 319-423.

[2] S. H. Kan,"*Metrics and Models in Software Quality Engineering*", Second Edition, Addison-Wesley, 2003.

[3] NIST2002, „*The Economic Impacts of Inadequate Infrastructure for Software Testing"*, National Institute of Standards and Technology, U.S. Department of Commerce, 2002.

[4] Lj. Lazić, N. Mastorakis, Cost Effective Software Test Metrics, WSEAS TRANSACTIONS on COMPUTERS , Issue 6, Volume 7, June 2008.

[5] Lj. Lazić, A. Kolašinac, Dž. Avdić. "The Software Quality Economics Model for Software Project Optimization", WSEAS TRANSACTIONS on COMPUTERS, Issue 1, Volume 8, p21-47, January 2009.

[6] Lj. Lazić, N. Mastorakis." OptimalSQM:Integrated and Optimized Software Quality Management", WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS, Issue 10, Volume 6, p p 1636-1664, ISSN: 1790-0832, October 2009.

[7] C. Jones, "*Software Assessments, Benchmarks, and Best Practices*", Addison-Wesley, 2000.

[8] D.F. Rico, "*ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*", J. Ross Publishing, Boca Raton, FL, 2004.