Cost/Benefit-Aspects of Software Quality Assurance

Master Seminar Software Quality

Marc Giombetti

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
giombett@in.tum.de

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 paper presents the cost estimation models COCOMO, its successor COCOMO II and COUALMO, which is a quality estimation model and has been derived from COCOMO II. Furthermore an analytical idealized model of defect detection techniques is presented. It provides a range of metrics: the return on investment rate (ROI) of software quality assurance for example. The method of ROI calculation is exemplified in this paper.
In conclusion an overview on the debate 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. Since all known models make heavy use of empirically gained data, it is very important to question their results judiciously and avoid misreadings.

Table of Contents

1 Introduction

The usage of software is pervasive in our society and software has taken a central role in our daily business and private life. Software is used in planes, trains, cars, banking systems aso., and therefore the software's quality plays a crucial role. The quality is important for the acceptance of the software by the user and thus is a key factor to the success of the software product.

Software systems are expensive products because their construction involves a lot of skilled people. Companies which develop software often spend excessive amounts of money to get an high quality software, which overcomes the firms actual quality needs. On the other hand some companies do not take quality assurance seriously enough or do not spent enough money, or do not use the right techniques for the quality assurance of their software production. It has often been seen that companies let an immature software skip over to field production. A possible software failure may then lead to millions of breakdown costs, loss of reputation, loss of market shares or even injure people. Thus, it is important to find the right balance between quality and quality assurance costs. The available budget should be invested pareto-optimally into the right quality assurance techniques to get the appropriate quality given a certain budget.

This work focuses on the main questions of how software quality assurance can be applied economically. It will give an insight into software quality cost, it´s calculation and present some models which enable the selection of the appropriate amount of specific quality assurance techniques to find the best solution for the investment in quality assurance effort. An idealized model of defect-detection techniques will be presented and this model can be used as starting point to calculate different metrics as return on investment.

1.1 Software quality

With respect to software system quality, it is not always possible to achieve the "best quality", but the intension is to create a software system having the right quality for a given purpose.

As a matter of fact, it is important for each software development project to define its specific meaning of software quality during the planing phase. On the one hand the quality of a software is adequate if all the functional requirements are met. On the other hand the softwares quality is also defined over non-functional requirements as reliability, usability, efficiency, maintainability and portability. This set of characteristics is defined in the international standard for the evaluation of software product quality ISO/IEC 9126-1:2001 [ISO01]. For each characteristic there exists a set of sub-characteristics which all contribute to the software quality to some extend. A more detailed description of the ISO 9126 can be found in Chapter **??**: *Quality Requirements*.

As example one could look at the availability of software. For an office application, an availability of 99,9%, which corresponds to an average downtime of 8,76 hours/year, is fairly appropriate. In respect to availability this office application is of high quality. Contrary to this, a power plant control software, having the same availability of 99,9% which stands for an average downtime of 8,76 hours/year too, is definately not acceptable. An unsafe failure might result in a disaster, polluting the environment and possibly injuring people. This shows that it is not enough only to consider a certain metric of a quality characteristic, but that it is important to see the application environment of the software too. Additionally it is important to look at financial issues. Quality assurance is costly and the expenses to be made to achieve the right quality are of interest to the project management and the customer. It is necessary to estimate and measure software quality costs.

1.2 Software quality costs

We have introduced different types of software quality characteristics and mentioned that it is not always possible to achieve the best quality, but that it is important to get the right quality for a certain software. The reason why the achievement of the "best" software quality is not possible, is mainly a financial issue. The higher the software's quality, the higher the quality assurance costs. Unfortunately the relation between the software quality improvement and the quality assurance investment effort is not linear. Todays software engineers and project managers are more and more aware of the software costs incurring over the entire software lifecycle. It is of paramount importance to find the right trade off between the software development quality assurance costs and the possible costs which arise when the software fails at the client. This is the only way of veraciously handling all the costs of a software system.

In the last three decades, there has been a lot of scientific work on finding relationships between quality and cost of quality. *Quality costs* are the costs associated with preventing, finding and correcting defective work [Wag06]. The *Cost of Quality* approach is such a technique. Mainly it is an accounting technique that is useful to enable the 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 [Kra98]. But which relationships exist between quality and cost of quality?

According to J.M. Juran and F.M. Gryna´s *Juran's Quality Control Handbook* [JF88] as well as P. Crosby´s book *Quality Is Free* [Cro80], the cost of quality can be subdivided into two categories: *Conformance* and *nonconformance* costs. Figure 1 gives an overview on the quality and how it relates to different types of costs.

Conformance costs, also known as control costs, can be partitioned into *prevention costs* and *appraisal costs*. Prevention costs include money spend on quality assurance, so that the software meets its quality requirements. Prevention

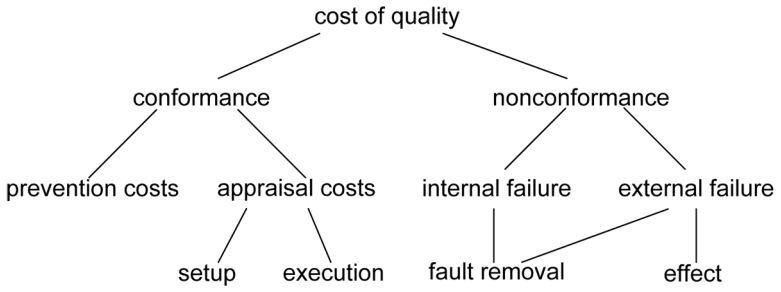

Fig. 1. Overview over the costs related to quality [JF88] - Extension of S. Wagner [Wag06]

costs for example are tasks like training, code and requirements reviews, tool costs and quality audits. All these quality assurance activities *prevent* the injection of various types of faults. The appraisal costs emerge from activities like developing test cases, reviews, test data and executing the test cases. *Setup* and *execution* costs are the most common appraisal costs. Setup costs cover all initial costs for configuring the development environment, acquiring licenses for test tools, aso. The execution costs cover all the costs which arise during the actual test-runs, review meetings aso. The execution costs are mainly personnel costs.

Nonconformance costs, also known as *failure of control* costs, emerge when the software does not meet the defined quality requirements. Nonconformance costs exist in two forms: *Internal failure* costs and *external failure* costs. The first type contains all the costs that arise due to a misfunction of the software during development and before the final delivery to the customer. The second type contains the costs that result from a failure of the software at the customer. After the establishment of a test process the internal failure costs increase and at the same time the external failure costs go down. There is an interdependency between both types of costs.

The nonconformance costs contain *fault removal* and *effect* costs. The fault removal costs are linked to the internal failure, as well as the external failure costs. This means that removal costs arise if a failure is detected internally as well as if the software fails at the customer. The last important type of costs are the *effect* costs. Effect costs arise when the software fails externally. They include all the costs that are caused by the software failing at the customer. Failure costs are not part of the effect costs. Examples for effect costs are loss of sales because of reputation, law costs, loss adjustment payments aso. [Wag06]

Up to now we have seen which types of quality costs exist and we will proceed by taking a look at how quality and software costs in general, can be estimated.

2 Quality and cost estimation

Cost estimation models are not new to the software industry, and today there exists a whole set of cost estimation models. Cost estimation is important because it removes some of the uncertainty in respect to the required expenditures. The objective is to make the best possible estimation of the costs, given a set of project factors and the skills of the development team. On the one hand companies need good cost estimates to be competitive in the market and to win call for bids. On the other hand it is important for them not to underestimate costs, because if they offer projects for a fixed price, they will narrow profits or even lose money. One of the first software cost estimation models has been developed by Barry Boehm in the 1970s. In his book *Software Engineering Economics* [Boe81] Boehm presents the COnstructive COst MOdel (COCOMO).

Quality estimation is a bit more complex and requires more advanced models. Whereas costs can be measured, it is more difficult to measure quality. The COnstructive QUALity MOdel COQUALMO is an extension to COCOMO and aims at determinating the software quality. Because the quality of a software product is directly related to the number of residual defects in the software, COCOMO takes the approach of predicting the quality of the software by estimating the the number of residual defects per/KDSI (Thousand of Source Lines of Code).

2.1 COCOMO & COCOMO II

In the following we introduce COCOMO as well as COCOMO II and provide some information on these techniques because they constitute the basis for COQUALMO. COQUALMO is a software quality estimation model and this work will mainly focus on this technique as an example for a quality estimation model. COQUALMO will be presented in detail in Section 2.2.

The main feature of COCOMO is to predict the required effort for a software project. Boehm developed COCOMO empirically by running a study of 63 software development projects and statistically analyzing their their project characteristics, people skills, performance and invested effort. The output of the COCOMO model is an effort prediction for the software development expressed in months. Because the main software development cost driver is the developer activity and the resulting personnel costs, one can assume that cost and effort are nearly the same.

To predict the effort the following equation is used:

$$\text{EFFORT} = a \cdot (\text{KDSI})^b$$

The constants a and b vary according to the type of project. KDSI is a measure to determine the software size, namely the Kilo Delivered Source Instructions. COCOMO distinguishes between three different development modes to enable more accurate predictions:

- *Organic:* Projects are small and similar to previous projects and are developed in a stable and familiar environment.
- *Semi-detached:* Between organic and embedded mode.
- *Embedded:* Projects have tight an inflexible requirements and constraints and require a lot of innovation.

Table 1 contains the effort equations for the different development modes. As intuitively expected: with increasing complexity of the project, the parameters a and b increase and thus the required effort increases too.

Development mode:	organic	$\text{EFFORT} = 2.4 \cdot (\text{KDSI})^{1.05}$
	semi-detached	$\text{EFFORT} = 3.0 \cdot (\text{KDSI})^{1.12}$
	embedded	$\text{EFFORT} = 3.6 \cdot (\text{KDSI})^{1.20}$

Table 1. COCOMO effort equations for different development modes.

We have now seen COCOMO in its most basic form: *the basic model*. There also exists an *intermediate model* and a *detailed model* which use an *effort adjustment factor (EAF)* which is multiplied with the effort calculation to get more accurate results. The EAF is the product of 15 cost factors subdivided into four categories: *platform costs*, *product costs*, *personnel cost* and *project costs*. In the intermediate model the following equation is used to determine the effort:

$$\text{EFFORT} = a \cdot \text{EAF} \cdot (\text{KDSI})^b$$

We will not go into further detail on how to use COCOMO, because the first version is outdated and a lot of improvements have been made meanwhile. The interested reader may refer to *Software Engineering Economics* [Boe81] for additional information on the first version of COCOMO.

The second version of COCOMO was developed in the 1990s and is mainly an adjustment of the first version to the modern development lifecycles and to the new techniques in software development. COCOMO II has been calibrated using a broader set of empirically collected project data, and in contrast to COCOMO additionally focuses on issues as:

- Non-sequential and rapid-development process models.
- Reuse driven approaches involving COTS packages, reengineering, application composition and application generation capabilities.
- Object oriented approaches supported by distributed middleware.
- Software process maturity effects and process-driven quality estimation.

Furthermore, COCOMO II now contains a set of new cost drivers. These cost drivers are subdivided into four categories: platform, product personnel and project cost drivers. Table 2 contains a list of these cost drivers, which are

Category	Cost driver
Platform	Required Software Reliability (RELY) Data Base Size (DATA) Required Reusability (RUSE) Documentation Match to Life-Cycle Needs (DOCU) Product Complexity (CPLX)
Product	Execution Time Constraint (TIME) Main Storage Constraint (STOR) Platform Volatility (PVOL)
Personnel	Analyst Capability (ACAP) Programmer Capability (PCAP) Applications Experience (AEXP) Platform Experience (PEXP) Language and Tool Experience (LTEX) Personnel Continuity (PCON)
Project	Use of Software Tools (TOOL) Multisite Development (SITE) Required Development Schedule (SCED) Disciplined Methods (DISC) Precedentedness (PREC) Architecture/Risk Resolution (RESL) Team Cohesion (TEAM) Process Maturity (PMAT)

Table 2. COCOMO II cost drivers [CB99].

important and will be used as *defect introduction drivers* in COQUALMO in Section 2.2.

Both COCOMO and COCOMO II models make use of empirically collected data and are only as good as the accuracy of this data. The quality of the empirical data used to calibrate the model has a direct influence on the quality of the estimation outcome of the model. Estimations by definition tend to be subjective and should always be looked upon with the necessary skepticism. For example the constants a and b are not fixed by the model, but every software company should adjust them based on the experience they gain from their daily software projects. To make the models as useful as possible, as much data as possible should be collected from projects and used to refine the model. A good way to store and learn from daily project data, is the application of the *Experience Factory* approach proposed by Basili, Caldiera and Rombach [BCR94].

COCOMO as well as COCOMO II do not make predictions on the quality of the software, neither which quality assurance techniques should be used to get the right quality at *optimized* costs. Quality is directly related to the number of defects which reside in the software. Intuitively, the more defects there are in the

software, the poorer quality is. Therefore it is of interest to have a model which quantifies the number of defects that get into the software as well as the number of defects that are removed from the software. COQUALMO is one instance of such a model and will be presented in the following:

2.2 COQUALMO

The COnstructive QUALity MOdel COQUALMO is an extension to the COCOMO II model. It determines the rates at which software requirements, design, and code defects are introduced into a software as a function of calibrated baseline rates, modified by multipliers determined from the project´s COCOMO II product, platform, people and project attribute ratings [BHJM04]. It enables 'what-if' analyzes that demonstrate the impact of various defect removal techniques and the effects of these attributes on software quality. It additionally provides insights into determining shipment time, assessment of payoffs for quality investments and understanding of interactions amongst quality strategies. Additionally it relates cost, schedule and quality of software. These characteristics are highly correlated factors in software development and form three sides of the same triangle. Beyond a certain (the "Quality is Free" point [Cro80]), it is difficult to increase the quality without increasing either the cost or schedule or both for the software under development [CB99].

With the development of COQUALMO Boehm aimed at facilitating the finding of a balance between cost, schedule and quality. Additionally to COCOMO II, COQUALMO is also based on the *The Software Defect Introduction and Removal Model*. The idea behind this model is that defects conceptually flow into a holding tank through various defect source pipes. Basically this means that the defects made during the requirements analysis, the design, the coding aso., flow through the defect source pipes into the software. On the other side there are also defect removal pipes through which the defects removed by quality assurance activities (a.e. testing) conceptually flow out of the software again. In the following the *Defect Introduction Model* and the *Defect Removal Model* will be presented and the relations which exist between COCOMO and COCOMO II will be outlined.

Defect Introduction Model: In COQUALMO, defects are classified based on the origin they result from. There exist *requirements defects*, *design defects* and *code defects*. The purpose of the Defect Introduction (DI) model is to determine the number of non-trivial requirements design and coding defects introduced into the software during development. As input to the DI model an estimation of the software size is necessary. This estimation may be thousand source lines of code (KDSI) and/or function points. Furthermore COQUALMO requires 21 (Disciplined Methods DISC is left out) of the 22 multiplicative DI-drivers of COCOMO II (see Table 2) as input data. The usage of the COCOMO II drivers not only makes the integration of COCOMO II into COQUALMO straight forward,

but also simplifies the data collection activity and the model calibration which have already been setup for COCOMO II.

There exist three categories of severity of defects: *critical*, *high* and *medium*. To actually calculate the total number of defects introduced into the software the following formula is used:

$$\text{Number of defects introduced} = \sum_{j=1}^{3} A_j (\text{Size})^{B_j} * \prod_{i=1}^{21} (\text{DI driver})_{ij}$$

where j identifies the three artifact types (requirements, design, code) and A is a multiplicative constant which is determined experimentally. $Size$ is the project size in KDSI or FP. B is initially set to 1 and accounts for economics of scale. Further details on the calibration of the model and the usage of parameter B can be found in *Modeling Software Defect Introduction and Removal: COQUALMO* [CB99]. Now that we can quantify the defects that get into the software, we look at the model of how they get out again.

Defect Removal Model: The Defect Removal (DR) model is a post-processor to the DI model. The concept main feature of DR model is to estimate the number of defects removed from the software by certain quality assurance activities.

These activities include three profiles: *Automated Analysis*, *People Reviews* and *Execution Testing and Tools*. Each of these profiles has different levels of increasing defect removal effectiveness from *very low* to *very high*. Table 3 contains the necessary defect removal investment and the rating scales for the mentioned profiles. The rating scales are the rows of the matrix whereas the activity profiles form the columns. The table then imposes what has to be done in each profile to achieve a certain rating level of quality. Additionally to every level of each of these profile, *Defect Removal Fractions* (DRF) are associated. These fractions are numerical estimates and have been determined by experts in a two-round Delphi estimation session. The interested reader can find information on the Delphi estimation process in [SG05].

As input the COQUALMO DR model requires the number of non-trivial *requirement, design and coding defects introduced* (= the output of the DI model). Furthermore the defect removal profile levels as well as the software size estimation are mandatory input parameters for the DR model. The model outputs the number of residual defects per KDSI (or per Function Point). The following example should give you a better feeling for the measure. Figure 2 shows a chart of the COQUALMO estimated delivered defect densities for the different defect removal rating categories. The chart is based on values of a calibrated baseline which have been rounded slightly to simplify the handling and to avoid an overfitting of the model. The rounded data contains 10 requirements defects, 20 design defects and 30 code defects for the *Very low* removal rating. One can see that for a *Very low* defect removal rating, 60 delivered defects are left in

Rating	Automated analysis	Peer reviews	Execution testing and tools
Very low	Simple compiler syntax checking	No peer review	No testing
Low	Basic compiler capabilities	Ad hoc informal walkthroughs	Ad hoc testing and debugging
Nominal	Compiler extension Basic requirements and design consistency	Well-defined sequence of preparation, review, and minimal follow-up	Basic test, test data management, problem tracking support; Test criteria based on checklists
High	Intermediate-level module and inter-module; Simple requirements and design	Formal review roles with well-trained participants, basic checklists, and follow-up	Well-defined test sequence tailored to organization; Basic test-coverage tools and test support system; Basic test process management
Very high	More elaborate requirements and design; Basic distributed-processing and temporal analysis, model checking and symbolic execution	Basic review checklists and root-cause analysis; Formal follow-up using historical data on inspection rate, preparation rate, and fault density	More advanced test tools, test data preparation, basic test oracle support, distributed monitoring and analysis, and assertion checking; Metrics-based test process management
Extra high	Formalized specification and verification; Advanced distributed processing	Formal review roles and procedures; Extensive review checklists and root-cause analysis. Continuous review-process improvement; Statistical process control	Highly advanced tools for test oracles, distributed monitoring and analysis, and assertion checking; Integration of automated analysis and test tools; Model-based test process management

Table 3. Defect-removal investment rating scales for COQUALMO [HB06].

the software. If more effort is spent and a *Very high* rating level is achieved, the delivered defect density is reduced to 1,6 delivered defects per KDSI.

The quality assurance team and the developers are mainly interested in the number of defects which reside in the software. The residual defects metric is important from a technical as well as a financial point of view, because every defect leading to a fault at the customer also leads to effect costs. The number of residual defects in artifact j is:

$$DRes_{Est,j} = C_j * DI_{Est,j} \prod_i (1 - DRF_{ij})$$

where C_j is a calibration constant, $DI_{Est,j}$ is the estimated number of defects of artifact type j introduced and i can take the values from 1 to 3 according to the type of DR profile (automated analysis, people reviews, execution testing and tools). The last variable DRF_{ij} is the Defect Removal Function for defect removal profile i and artifact type j. In the following we will highlight how COCOMO and COQUALMO are related.

Relationship between COCOMO II and COQUALMO: COQUALMO is integrated into COCOMO II and cannot be used without it. Figure 3 shows the DI model and the DR model which are integrated into an existing COCOMO II

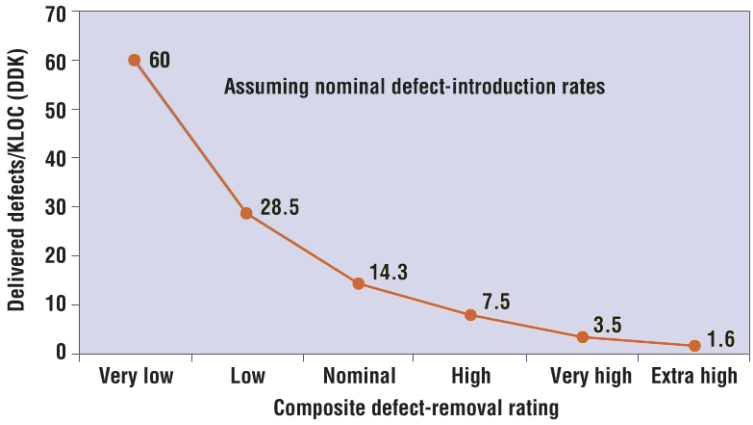

Fig. 2. Estimated delivered defect densities using COQUALMO [HB06]

cost, effort and schedule estimation. The dotted lines are the inputs and outputs of COCOMO II. Apart from the software size estimation and the platform, project, product and personnel attributes, the defect removal profile levels are necessary to predict the number of non-trivial residual requirements, design and code defects.

Because of this tight coupling between COCOMO II and COQUALMO, I think that a project manager should use COQUALMO for a software project if COCOMO II estimates already exists. The effort to implement COQUALMO is worth it, when considering the payoff a project can get from applying COQUALMO. Nevertheless there are also some drawbacks in the usage of COCOMO and COQUALMO. One big disadvantage of COCOMO is that is uses the Size (in KDSI) to calculate the effort. Because effort calculations are usually done in a very early project stage, when there is often not enough information to estimate the Size of the complete product. Also the weighting of the cost factors is not easy at this early point.

Additionally the data which is collected using COQUALMO can again be used to improve the estimates of the different sizing parameters of COQUALMO and COCOMO II. Furthermore this data is the foundation of further investigations on software quality and cost. Section 2.4 illustrates an example on how return on investment calculations for software projects can be accomplished. The ROI calculations are one example of a metric which can be based on the analytical model of defect-detection techniques which will be presented in the following.

2.3 An analytical model of defect-detection techniques

The following analytical model of defect-detection techniques has been developed by Stefan Wagner as part of his PhD-thesis on *Cost-Optimisation of Analytical*

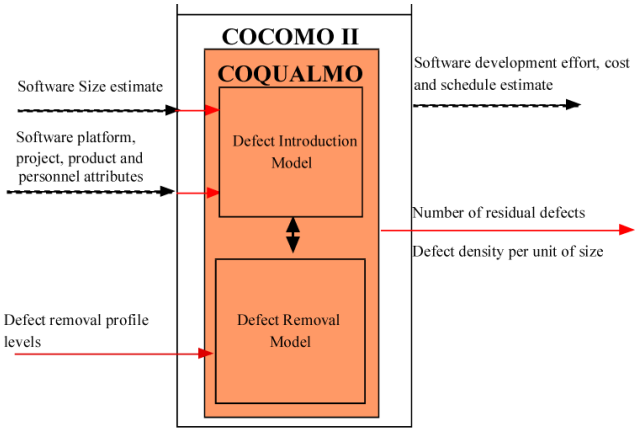

Fig. 3. The Cost/Schedule/Quality Model: COQUALMO Integrated with COCOMO II [CB99]

Software Quality Assurance [Wag07]. There are a few similar models, but this model was picked, because it clearly highlights the different cost components, precisely models the relations and is well documented. The model is a refinement and extension of the model by Collofello and Woodfield [CW89] which uses fewer input factors. It is a general model which can be used to analyze various types of quality assurance techniques. For example it can be used to analyze different types of testing or static analysis techniques to see which are most effective for a certain project. The model is a cost-based *ideal model of quality economics* and it doesn't focus on the use of the model in practice, but it is rather theoretical and mirrors the actual relationships as faithfully as possible.

Components: In the following we shall introduce the main components of the model, which is subdivided into three parts. All components are dependent on the spent effort t as global parameter.

- *Direct costs* $d(t)$, are costs which can be directly measured during the usage of a technique.
- *Future costs* $f(t)$, are the costs in the field which really incurred.
- *Revenues / saved costs* $r(t)$, are the costs in the field which could have emerged but which have been saved because failures were avoided due to quality assurance.

The model determines the expected values of these components, denoted by E. Before the introduction of the computation formulas, we will outline the model assumptions, to see which criteria have to be met to apply the model.

Assumptions: Because the model is an ideal model, the assumptions require an idealized environment:

- *The found faults are perfectly removed.* - This means that each fault which is found is removed and no new faults are introduced during the removal process.
- *The amount or duration of a technique can be freely varied.* - This is needed because there is a notion of time effort in the model to express for how long and with how many people a technique has been applied.

It is clear that in practice these assumptions cannot hold most of the time, but they are meant to clearly simplify the model.

Difficulty: The model requires a further notation for the characterization of quality assurance techniques. The difficulty of an application of technique A to find a specific fault i is denoted by $\theta_A(i)$. In a mathematical sense, this is the probability that technique A does not defect default i. Furthermore we introduce t_A which is the length of the technique application A. The length is the effort (a.e. measured in staff days) that was spent for the application of a technique.

In the later formulas, the expression $1-\theta_A(i, t_A)$ is the probability that a fault is at least detected once by technique A. Furthermore the model requires the concept of *defect classes* which regroups several defects according to the type of document they are contained in. For every defect there exists a document class (e.g. requirements defects or code defects), too. This subdivision in classes is important because some type of defect removal activity is only applicable to a certain type of defects. For example, it generally doesn't make sense to apply a formal verification technique to find requirements defects.

Defect propagation: Defect propagation is another important aspect which has to be taken into consideration. For the defects occurring during development, we know that they are not (always) independent of each other. One of the existing dependencies is the propagation of defects over the different phases of the software development process. The different phases are not considered, but the model rather takes the different artifact types and documents into consideration.

One defect in a certain artifact can lead to no, one or more defects in later documents. Figure 4 illustrates how defects may propagate over documents. In this example requirements defects lead to several design defects and a test specification defect. The design defects again propagate to code defects which can entail further test specification defects.

For each document type c, I_c is the set of defects in c and the total set of defects is $I = \bigcup I_c$. Additionally each defect has a set of predecessor defects named R_i, which may be the empty set. The usage of predecessors is important, because a defect can only be present in one artifact if none of the predecessors has already found it.

Model components: In the following the formulas for the expected values of the direct costs, the future costs and the revenues are given. Later the formulas for the combination of the different techniques will be given too. For the sake

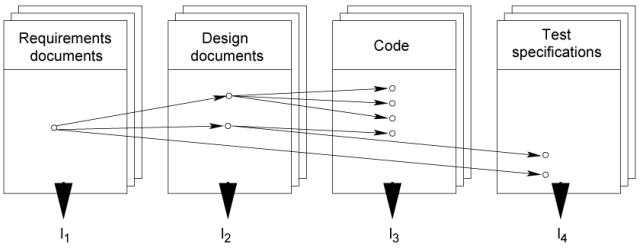

Fig. 4. Defect propagation over documents [Wag07].

of simplicity, the defect propagation is not taken into consideration in the following equations. Nevertheless the defect propagation was introduced previously because it plays a main role in practice as well as in the ideal model. Presenting the defect propagation issue would go beyond the scope of this paper, therefore we will only introduce the simple equations. Details on the model can be found in [Wag07].

Direct costs: *The direct costs are those costs that can be directly measured from the application of a defect-detection technique.* To determine the expected value for the direct costs $E[d_A(t_A)]$ the following formula is used:

$$E[d_A(t_A)] = u_A + e_A(t_A) + \sum_i (1 - \theta_a(i, t_A))v_A(i) \tag{1}$$

where u_A are the setup costs, $e_A(t_A)$ are the execution costs, $v_A(i)$ are the fault removal costs of the technique A and $1 - \theta_A(i, t_A)$ is the probability that a fault is at least detected once by technique A.

Future costs: The future costs are the costs which emerge in case some defects are not found. As introduced in Section 1.2, these costs can be subdivided into fault removal costs in the field $v_F(i)$ and failure effect costs $c_F(i)$. The equation to determine the expected value of future costs $E[f_A(t_A)]$ is the following:

$$E[f_A(t_A)] = \sum_i \pi_i \theta_A(i, t_A)(v_F(i) + c_F(i)) \tag{2}$$

where
$\pi_i = P$(fault i is activated by randomly selected input and is detected and fixed).

Revenues: Revenues are saved future costs. The cost categories are the same as for the future costs, but now we are looking at the defects that we find instead of the ones we miss. The equation to determine the expected value of revenues

$E[r_A(t_A)]$ is the following:

$$E[r_A(t_A)] = \sum_i \pi_i(1 - \theta_A(i, t_A))(v_F(i) + c_F(i)) \tag{3}$$

Combination: up to now we have seen the equations for the different types of expected costs, which were always defined for one technique of quality assurance. Nevertheless, more than one technique to find defects is used in practice. The reason for this is that different techniques find different defects. The model takes this into consideration and allows the combination of different techniques. To formalize this, the model defines X as the ordered set of the applied defect detection techniques. To get the total direct costs, you have to sum over all technique applications X. Then we use Formula 1 and extend it that not only the probability that the technique finds the fault is taken into account, but also that the predecessor techniques have not found the fault. The predecessor defects R_i have to be taken into consideration too. To improve readability, we use

$$\Theta(x, i) = \prod_{y<x} \left[\theta_y(i, t_y) \prod_{j \in R_i} \theta_y(j, t_y) \right] \tag{4}$$

as the probability that a fault and its predecessors have not been found by previous (before x) applications of defect detection techniques. Then for each technique y that is applied before x, the difficulty for the fault i and all its predecessors in the set R_i is multiplied. The following formula is then used to determine the expected value of the combined direct costs d_X of a sequence of defect-detection technique applications X:

$$E[d_X(t_X)] = \sum_{x \in X} \left[u_x + e_x(t_x) + \sum_i \left((1 - \theta_x(i, t_x)) \Theta(x, i) \right) v_x(i) \right] \tag{5}$$

In the same way we can introduce the formula for the expected value of the revenues of several technique applications X:

$$E[r_X(t_X)] = \sum_{x \in X} \sum_i \left[(\pi_i (1 - \theta_x(i, t_x))\, \Theta(x, i))(v_F(i) + c_F(i)) \right] \tag{6}$$

Here we consider the faults that actually occur, and that are detected by a technique. Additionally neither itself not its predecessors have been detected by previously applied techniques.

The expected value of the total future costs are the costs of each fault with the probability that the fault actually occurs and that all the techniques and the predecessors failed in detecting it. It can be determined using the following

formula:

$$E[f_X(t_X)] = \sum_i \left[\pi_i \prod_{x \in X} \theta_x(i, t_x) \underbrace{\prod_{y<x\ j} \prod_{in R_i} \theta_y(j, t_y)}_{\Theta'(x,i)} (v_F(i) + c_F(i)) \right] \quad (7)$$

where $\Theta'(x, i)$ is similar to Formula 4 but only describes the product of the difficulties of detecting the predecessors of i. In this case the probability if the predecessor has actually detected the fault is not necessary.

With the definition of the model components, it is now possible to calculate some economical metrics of the software quality assurance process.

Return On Investment: The return on investment ROI is a well known metric from economics. In economics the ROI is commonly defined as the gain divided by the used capital. In this case the rate of return for a specific defect-detection technique is of interest. We can calculate the ROI as follows:

$$\text{ROI} = \frac{\text{profit}}{\text{total cost}} = \frac{r_x - d_x - f_x}{d_x + f_x} \quad (8)$$

The profit is equal to the revenue minus the direct and future costs. The total cost is the sum of the direct and the future cost.

Return on investment can be an assessment of whether the investment in a specific defect-detection technique is justified by the quality improvement and the resulting cost reduction over the entire lifecycle. It sometimes even makes sense to have a negative ROI, when there is a very high risk to human life or the environment, or when there are important customer relationships which should not be jeopardised. It is even possible to express factors like loss of human life or the loss of a customer as costs. However this costs are often very difficult to quantify and in some cases legal issues may prevent this. It seems ethically not correct to express the loss of a human life because of a defect in a safety critical software system as effect costs. Nevertheless, theoretically there is no hindrance why it should not be possible to include such injury effort costs into the total costs of a software.

2.4 Example for a return on software quality investment calculation

In this section an example of how ROI calculations can be made in practice will be given. This example is part of the *The ROI of Software Dependability - The iDAVE Model* article by Barry Boehm et al. [BHJM04] and has been slightly adapted for this paper. The example focuses on two completely different software systems: The order-processing system of a mountain bike manufacturer and the NASA Planetary Rover. The intention is to determine the ROI of quality

investments in availability for both system. Both systems have high availability requirements, but the project characteristics and the breakdown costs are completely different. In case of unavailability of the order-processing system, the mountain bike manufacturer will not be able to sell mountain bikes during the downtime. The unavailability of the NASA Planetary Rover would imply the end of it´s mission on a foreign planet, because it wouldn't be able to transmit it status to earth, neither be controllable from the command center anymore.

Return on investment calculations using iDAVE: The ROI analysis of this example is made using the Information Dependability Attribute Value Estimation (iDAVE) model. The iDAVE model is strongly related to COCOMO II and is a derivative of COQUALMO. We will not present the intricacies of this model, but we will focus on a practical example which does not require detailed knowledge on the iDAVE model. *"The iDAVE dependability ROI analysis begins by analyzing the effect of increasing dependability investments from the nominal business levels to the next higher levels of investment in analysis tool support, peer-reviews practices and test thoroughness"* [BHJM04]. The different levels of investment are the same as the defect removal levels in COQUALMO (see Table 3). The effects of investments are coupled to the RELY attribute, which is the COCOMO II reliability attribute (see Table 2). In the example the ROI calculations of the order-processing system will be done first, and the calculations for the NASA Planetary Rover will follow in the second part.

Mountain bikes order-processing system: The RELY rating scale for both the mountain bikes system includes the *Nominal*, *High*, *Very high* and *Extra high* ratings. For each project and the respective ranking levels the availability which is defined as:

$$\text{Availability} = \frac{\text{Mean Time Between Failure}}{\text{Mean Time Between Failure} + \text{Mean Time To Repair}}$$

is determined. The values of the *Mean Time Between Failure* and the *Mean Time To Repair* have been estimated by business domain experts. Table 4 gives the complete dataset for this example including the rating scales, availability calculations, financial values and the resulting return on investment rates.

For the order-processing system availability will be used as a proxy for dependability. Then it can be assumed that a one percent increase in downtime is equivalent to a one percent loss in sales. The total expected loss value for a given availability are now used to calculate the *Increased Value* for changing from one rating level to the next higher one. In the case of the order-processing system an improvement from *Nominal* to *High* RELY rating (from availability 0.9901 to $0,9997$) leads to an Increased Value (rounded values) of:

$$0,01 \cdot (531M\$) - 0,0003 \cdot (531M\$) = \\ 5,31M\$ - 160K\$ = 5.15M\$$$

Project	RELY rating	MTBF (hrs.)	MTTR (hrs.)	Availability	Loss (M$)	Increased value (M$)	Cost (M$)	Change (M$)	ROI
Mountain bikes order-processing system	Nominal	300	3	0,9901	5,31	0	3,45	0	-
	High	10K	3	0,9997	0,16	5,15	3,79	0,344	14,0
	Very High	300K	3	0,99999	0,005	0,155	4,34	0,55	-0,72
	Extra High	1M	3	1	0	0,005	5,38	1,04	-1,0
NASA Planetary Rover	High	10K	150	0,9852	4,44	0	22	0	-
	Very High	300K	150	0,9995	0,15	4,29	25,2	3,2	0,32
	Extra High	1M	150	0,99985	0,045	0,105	31,2	6	-0,98

Table 4. Values for the ROI calculation example [BHJM04].

where 531M$ is the arithmetic mean of the sales per year (the possible loss). With this, the dependability ROI calculation leads to:

$$\text{ROI} = \frac{\text{Profit}}{\text{Cost}} = \frac{\text{Benefits} - \text{Cost}}{\text{Cost}} = \frac{5,15 - 0,344}{0,344} = 14,0$$

A related result is that the additional dependability investments have relatively little payoff, because the company can only save $5,31 - 5,15 = 0,16M\$$ by increasing the availability.

The results of this analysis initially were discussed with business experts of the mountain bike order-processing quality assurance team who pointed out that a negative ROI that resulted from the improvement of a *High* to a *Very High* RELY level would not let their interest in improvements disappear. As when the needs in availability are fulfilled, there emerge other important motivators. Reducing security risks is an example for a possible motivator. This explains why negative ROIs are no reason to stop quality assurance investments.

NASA Planetary Rover: The NASA Planetary Rover has higher needs to availability as the order-processing system, and thus at least requires a *High* RELY rating level. The values for the *Mean Time Between Failure* are equal to those of the order-processing system, but the *Mean Time to Repair* increases. According to the engineers at NASA, it roughly takes a week or 150 hours to diagnose a problem, prepare the recovery and test its validity. The main requirement of the Rover is survivability. In short terms, survivability means that the Rover always has to keep enough power and communication capacity to transmit its status to earth. In this example availability is used as a proxy for survivability, because it is more straightforward to analyze. For the ROI calculation in this study, the total price of the mission (300M$) has to be known. The mission´s software costs account for 20M$, which is 7 percent of the entire mission cost. The software costs have been determined for the *Nominal* COCOMO II software RELY rating. Same as for the order-processing system one percent decrease in availability leads to a one percent of loss on the mission value of 300M$.

The dependability ROI analysis is yet again used to measure the effect of increasing the RELY rating to the next higher level. The transition from a *High* to a *Very high* RELY rating level corresponds to a cost increase from 22M$ to 25,2M$ and an increase in availability from 10K to 300K hours (see Table 4). The transition from a *Very High* to *Extra High* RELY rating is a special case because it exceeds the normal COCOMO II cost rating scale. It requires the extended COCOMO II RELY range. The required investment to fulfill this improvement is 6M$ and the added dependability will result in a negative payoff (ROI = -0,98). Nevertheless the NASA responsible pointed out that a negative ROI is acceptable for them too, because it reduces the risk of mission failure, loss of reputation or even harm to human life.

Example Summary: To sum up, Figure 5 compares the ROI analysis results of the mountain bike order-processing system and the NASA Planetary Rover. The chart summarizes Table 4 and points out how the return on investment rates change, while improving from one to the next higher level of reliability. It is clear that different projects have different break even points for the ROI rates. To conclude, there unfortunately is no recipe in when to stop quality assurance investments.

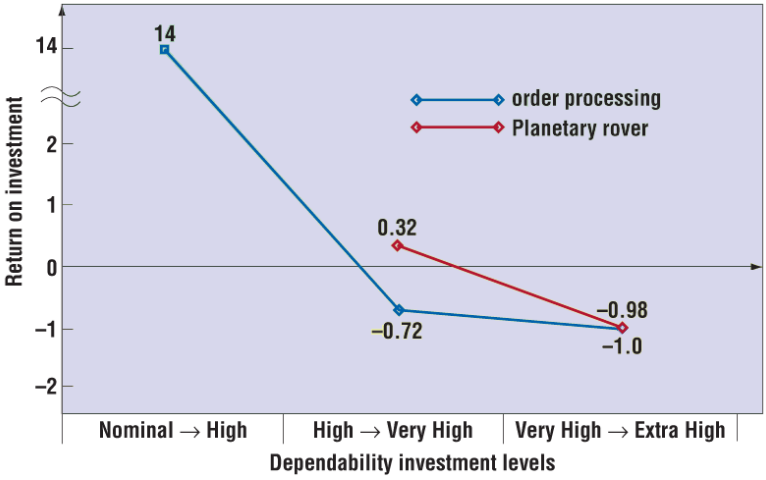

Fig. 5. Comparison of the ROI analysis for the mountain bike order-processing and the NASA's Planetary Rover [BHJM04].

3 Reflections on Cost-Effectiveness and Quality

In the following section I will make some deliberations on cost effectiveness and the quality of the presented models, as well as on the quality of the software product outcomes. This section reflects my personal opinion on those subjects. The first thing to be treated is the quality of the model calibration:

3.1 Model calibration

The different models that have been presented all make heavy use of empirical data. The problem with empirical data is that even though it might deliver good results for most of the application domains, it doesn´t have to be appropriate for a specific domain. The model output directly depends on the quality of the empirical calibration data. The empirical data might be project data, as well as data collected from experts (expert opinion). There exist several statistical techniques to determine the quality of empirical data. An example on checking the validity of the calibration of models based on empirical data can be found in Freimut et al. [FBV05]. A detailed study on different model calibration techniques was only mentioned for completeness reasons and would exceed the scope of this paper.

Nevertheless it is interesting to take a look at other problems related to the collection process and studies based on empirical data. One keyword in this context is *replicatability*. As already mentioned, results which are achieved in one specific software engineering domain, might be completely useless in another:

To illustrate how difficult the collection of empirical data is, and to see that even though the studies have been undertaken very precisely the results of two studies are completely different, it is interesting to take a look at *On the difficulty of replicating human subjects studies in software engineering* [LAEW08]. Even though this study is not about quality engineering, it shows how replications of the same experiences under equivalent conditions can lead to different implications. I picked this example because it is striking, well documented an reveals the difficulties of empirical software engineering and data collection.

The replication of results in software engineering is difficult, because it involves a lot of people having personal characteristics and a lot of influence factors on these people. In the original experience *The Camel Has Two Humps* undertaken by S. Dehnadi and R. Bornat, they claimed to have developed a test, administered before students were exposed to instructional programming material, that is able to accurately predict which students would succeed in an introductionary programming course and which would struggle. [DB06]. The authors of the paper set their sights in very carefully replicating a well documented experience to see if their results would agree or disagree with the original's. In either case, they would increase or decrease the confidence in the original hypothesis that *the camel has to humps*, so as to say that there are only the very good and very bad programmers, but no average programmers. Even though the replication was very precisely undertaken, the authors didn't achieve the same results

as for the original experience. The authors tried to find out what could have lead to the different results, and they conclude that there are so many human and external factors, as that it is nearly impossible to replicate an experience and obtain the same results. For this paper, the previous replication experience shows that even though the model calibrations as well as the execution of the quality estimates might have been done very carefully, the models nevertheless may lead to untrustworthy results.

Same as for the model calibration, it is also interesting to reflect about quality-engineering concerns:

3.2 Quality-Engineering concerns

The models that were presented in this work are good to estimate some aspects of the software quality, which are easy to measure. Unfortunately most quality attributes are not easily measurable. The models for example do not consider the costs which result from a lack of maintainability or usability of the software. For the availability of a software system the case is "simple" because availability is an absolute measure. There is a relationship between increasing the quality assurance investments and the resulting availability increase which can be measured and validated.

Changeability: Software systems are subject to change. External influences as changing regulations, laws or simply changed market conditions require adaptations to the software. It the software system is unmaintainable for any reason, the emerging costs will be enormous, because the software has to be rewritten from scratch. It might even be worse: the software cannot be rewritten because the development would take far to long and exceed the companies budget. This scenario would be a complete disaster for the company and it might be entangled in lawsuits, because its software is involuntarily violating laws and regulations. To continue, I would also like to point out another interesting aspect to software, which is the *Total Cost of Ownership* in respect to quality.

Total Cost of Ownership: The presented models mainly took software aspects into consideration. Nevertheless the cost-effectiveness of a system also depends on hardware costs. Additionally power consumption in data centers to actually run the software cannot be neglected. In times of *Green IT* and increasing electricity fees, it is also important to include the power consumption by servers and cooling into calculation. One quality attribute of software (running on a server) could be it´s power consumption per hour. Therefore server-software is of good quality if it can be virtualized, thus reducing the number of servers, rack space, power consumption and costly management effort. This might sound awkward at first, but this could play an important role in the years to come.

Usability: On an even higher level, companies aiming at reaching more accurate ROI of software quality estimates should think about how many employees

have to work with the software each and every day, and how frustrating the usage of the software might be. An effective, intuitive and easily usable software is of paramount importance. For example if several dialogs take to long to load, or are complicated to oversee, expensive working hours will be wasted for years each and every day. It is not neglectable that enterprise systems are expensive strategic investments, and thus will not seldom be used for a decade or even longer. As a matter of fact, different generations of employees will have to work with the software and will need to attend courses to learn how to handle it. The education costs are inversely related to the usability of the software. It is clear that if the software is to complex and cumbersome, the learning process slows down and takes longer, thus reducing the effectiveness of employees.

In my opinion it is important that in the future there should be more research and investigations to really capture all the influences software quality engineering has, thus enabling more precise rate of return calculations for software quality assurance.

4 Conclusion and outlook

The different cost estimation models presented in this work are the fundamental to the process minimizing, the total cost of software quality. But it is not sufficient that the software engineers are aware and know how to use such models. In contrary it is important to have a management team which is smart enough to look at the cost of quality over the entire lifecycle of the software product. It doesn't make sense to release immature software, just to reduce the short time expenses generated by quality assurance effort. The expenditure due to long term external failure in the field, will surely exceed the amount of saved short-term investments for quality assurance. Having an far-sighted and supportive management is of great necessity, but it is not a wild-card for achieving satisfactory return on software quality assurance investments. It is rather of paramount importance to know how to effectively make use of the presented models and always critically face the outcomes.

The presented models are a solid basis for companies to start their software quality assurance investments, but they are no push-button techniques. Software companies should always keep in mind the importance of the collection and appraisal of empirical data. They should see their companies as mature learning companies, and have robust experience factories where they save the data generated during the development process of their software projects [BCR94]. The quality of the data in the experience base is crucial, because nearly all cost and quality estimation models make use of this data to predict cost and quality of future projects.

References

[BCR94] Basili, V.R. ; Caldiera, G. ; Rombach, H.D.: Experience Factory. In: *Encyclopedia of Software Engineering* 1 (1994), S. 469–476

[BHJM04] Boehm, B. ; Huang, L. ; Jain, A. ; Madachy, R.: The ROI of software dependability: The iDAVE model. In: *Software, IEEE* 21 (May-June 2004), Nr. 3, S. 54–61

[Boe81] Boehm, B.W.: *Software Engineering Economics*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1981

[CB99] Chulani, S. ; Boehm, B.: *Modeling Software Defect Introduction and Removal: COQUALMO*. 1999

[Cro80] Crosby, P.B.: *Quality Is Free: The Art of Making Quality Certain*. Mentor Books, 1980

[CW89] Collofello, J.S. ; Woodfield, S.N.: Evaluating the effectiveness of reliability-assurance techniques. In: *Journal of Systems and Software* 9 (1989), Nr. 3, S. 191–195

[DB06] Dehnadi, S. ; Bornat, R.: The camel has two humps (working title). (2006). http://www.cs.mdx.ac.uk/research/PhDArea/saeed

[FBV05] Freimut, B. ; Briand, L.C. ; Vollei, F.: Determining inspection cost-effectiveness by combining project data and expert opinion. In: *Software Engineering, IEEE Transactions on* 31 (Dec. 2005), Nr. 12, S. 1074–1092

[HB06] Huang, L.G. ; Boehm, B.: How Much Software Quality Investment Is Enough: A Value-Based Approach. In: *IEEE SOFTWARE* (2006), S. 88–95

[ISO01] ISO/IEC: *ISO/IEC 9126-1:2001 Software engineering - Product quality - Part 1: Quality model*. Geneva, Switzerland : International Standards Organization, 2001

[JF88] Juran, J.M. ; F.M., Gryna: *Juran's Quality Control Handbook*. McGraw-Hill, 1988

[Kra98] Krasner, H.: Using the Cost of Quality Approach for Software. In: *CrossTalk. The Journal of Defense Software Engineering* 11 (1998), Nr. 11, S. 6–11

[LAEW08] Lung, Jonathan ; Aranda, Jorge ; Easterbrook, Steve M. ; Wilson, Gregory V.: *On the difficulty of replicating human subjects studies in software engineering*. New York, NY, USA, 2008

[SG05] Stellman, A. ; Greene, J.: *Applied Software Project Management*. First Edition. O'Reilly Media, 2005

[Wag06] Wagner, S.: A model and sensitivity analysis of the quality economics of defect-detection techniques. In: *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA : ACM, 2006, S. 73–84

[Wag07] Wagner, S.: Cost-Optimisation of Analytical Software Quality Assurance. (2007)