Interval Quality: Relating Customer-Perceived Quality To Process Quality

Audris Mockus and David Weiss
Avaya Research
233 Mt Airy Rd
Basking Ridge, NJ 07920
{audris,weiss}@avaya.com

ABSTRACT

We investigate relationships among software quality measures commonly used to assess the value of a technology, and several aspects of customer perceived quality measured by Interval Quality (IQ): a novel measure of the probability that a customer will observe a failure within a certain interval after software release. We integrate information from development and customer support systems to compare defect density measures and IQ for six releases of a major telecommunications system. We find a surprising negative relationship between the traditional defect density and IQ. The four years of use in several large telecommunication products demonstrates how a software organization can control customer perceived quality not just during development and verification, but also during deployment by changing the release rate strategy and by increasing the resources to correct field problems rapidly. Such adaptive behavior can compensate for the variations in defect density between major and minor releases.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software quality assurance*

General Terms

Measurement, Reliability

Keywords

Software Metrics

1. INTRODUCTION

A lingering question for software development organizations is how to evaluate what the customers' perception of the quality of a software product is, particularly how customer-perceived quality is related to observed in-process quality. An organization that makes changes in its development process in the expectation that software quality as observed during development will improve, would especially like to know that such in-process improvement will herald improvement from the customer viewpoint.

Correlating in-process improvement with customer perceptions has traditionally been a difficult problem. Success in solving this problem could lead to ways of predicting customer satisfaction with a product release, as well as ways of suggesting what kinds of process improvements are most likely to lead to improved customer satisfaction.

Our goal is to find a satisfactory way to quantify customers' perception of software quality in order to assess how process improvements lead to better user experiences. Our main focus is to find a practical approach to measure customers' perception of software quality, implement it in an organization, and validate its performance.

The major transformation in telecommunications industry towards IP telephony has increased customers' and solution providers' concern about the quality and reliability of the new platforms. Would they provide quality comparable to the traditional Time Division Multiplex (TDM) solutions? The impetus for this work was the perception of quality experts at Avaya's software development organization that the traditional quality measures do not match users' perception of quality and, therefore, they can not be meaningfully used to assess the quality of the new IP platforms. The subsequent development and analysis of the new measure provides some insights on why such dissatisfaction may occur.

We introduce an operationalization used in Avaya of a measure that is based on the probability that a user observes an adverse event within several months of installing software. We investigate its relationship with better known process and product measures [18, 9, 8], such as defect density and mean time between failures (MTBF). We also investigate the relationship of these measures to changing business and management priorities.

We call our measure "interval quality" (IQ) because it is based on observing failures during specific intervals after a product has been released to customers. The letter "I" may also stand for "Initial" or "Installation" because the interval in question starts at the time of system installation. We find surprising anti-correlation between several traditional quality measures and the probability that a customer will observe an adverse event. On the other hand, we observe that a shift in business and management priorities, such as focusing on quality rather than time to market, changed in line with that probability, as did the MTBF.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

We believe that the discrepancy may be at least partly explained by the fact that software quality can be controlled not just during development and verification, but also in early introduction (alpha and beta) phases and, most importantly, during deployment.

Organizations generally have a fixed set of resources to use for developing software and must make trade-offs about what techniques and tools to use to achieve their quality objectives prior to delivering the software to customers. However, as previously shown, factors outside of the development process also affect customer's perception of quality [17]. The deployment strategy used after the software is declared fit to be delivered to customers is one such factor. In particular, larger major releases are deployed more gradually, affecting fewer customers and, therefore, raising relatively fewer issues than would be expected based on release size alone. Minor releases are deployed more rapidly, affecting larger numbers of customers, and, therefore, raise more issues than would be expected based on their small size.

Our primary contribution is IQ, a practical new measure of software quality validated through its development and use in a large software organization and the investigation of its relationships with other quality measures, such as defect density. Our observational study of the impact of process on product quality finds differences between the behavior of these measures and IQ.

We present a variety of quality measures calculated from historical observations in order to facilitate further analysis by other researchers and comparison with other products. We also describe the nature and possible causes of the discrepancy and discuss the most appropriate measures of customer perceived quality.

We first provide the context for the study in Section 3, then we present the new measure of software quality in Section 4. In Section 5 we compare the quality measures and find dramatic differences. Validity issues are discussed in Section 6. We briefly summarize related work in 2 and conclude in 7.

2. RELATED WORK

Related work includes a number of studies of the characteristics of source code files with high fault potential. Many studies have used several *product measures*, measurements of a snapshot of the code itself, as predictors of fault likelihood. Code size (lines of code) is the canonical fault predictor. Measures of code complexity, such as McCabe's cyclomatic complexity [13] and Halstead's program volume [11] are other examples of product measures. There are a number of empirical studies [26, 2, 24, 21, 25, 19, 1] of product measures and fault rates.

Another class of measures for modeling fault rates are based on data taken from the change and defect history of the program. [28, 10] finds variables in the change history of collections of files, such as numbers of changes to those files and the average age of the lines in those files, which account in part for the number of faults observed to affect those files at a later time. COQUALMO [5] project size metrics and various process (development process) factors predict defect occurrences.

In contrast to the studies above, the work in [15] predicts the probability of failure pertaining to a *change* of a software entity, rather than to attempt to identify the probability of failure or the number of failures for a software entity.

A different line of work can be found in the *software reliability* literature. Here, one estimates the number of faults remaining in a fixed software system, in order to predict how many faults will be observed in a future time interval, assuming that the software does not continue to be changed. See [20, 12, 23, 18, 7, 4].

In our previous work [17] we modeled the probability that a user observes an adverse event and found that deployment predictors have an enormous effect on such a measure.

The IQ measure attempts to unify many of the ideas previously put forth in measuring and analyzing changes and measuring faults and failures in a way that can be used to manage industrial software development.

3. BACKGROUND

To illustrate the context of the study we present relevant aspects of the project and of the customer support process.

3.1 The customer support and Tier IV systems

Avaya uses a tiered support process. Lower tiers are handled in the service organization while Tier IV represents a very small fraction of issues that could not be resolved in lower tiers and that are escalated to the development organization. We have described the trouble ticket system and process in [17], here we focus on Tier IV and the development systems and process as described above.

Tier IV tickets are routed to various groups specializing in different products. More specifically, an issue raised against a product may represent a defect in a different, interoperating product. In such situations, the ticket is routed to the most relevant group in Tier IV. We use the group information to exclude tickets not related to the products in question.

For the releases we analyzed in developing IQ there are approximately 3,000 Tier IV trouble tickets. They affect about 60,000 systems in the field, out of about 4 million listed in Avaya's equipment database.

3.2 The software project

We examine the call processing software installed on many Avaya telephony systems. This software system is an established product and embodies several decades of knowledge and experience in the telephony field. In a recent release, the software contains approximately seven million lines of code mostly in C and C++. The software development organization deploys major releases on a fixed schedule, with subsequent minor releases that bundle patches and refinements to the system.

Many releases are in the field and are used by tens of thousands of customers, many of whose businesses depend on the high availability of the product. This makes the software exceedingly difficult to enhance while maintaining the smooth operation of the hardware/software combinations deployed.

Three primary systems contain information about the development of the software. The Modification Requests (MRs) tracking system and a version control system allow us to identify and measure the size of the software releases and numbers of defects and customer related defects. A Tier IV issue tracking system contains customer problems escalated to Tier IV, i.e., issues that could not be resolved by lower tiers in a service organization and that were likely to require software changes. Therefore they were escalated to Tier IV, which is part of the software development organization.

In addition, we use service and deployment systems that track, among other things, information about the time of installation and upgrades by customers. We use them to obtain the number of customers installing the releases and the dates when these installations occur.

3.2.1 The software change process

When a change to the software system is needed, a work item is created. Work items range in size from very large work items, such as releases, to very small changes, such as a single delta (modification) to a file. Figure 1 shows the organization of changes and data attributes associated with them in a typical change management system.

The project we consider here employs a version control system (VCS), which maintains versions of the source code and documentation, and a change request management system (CMS) that keeps track of individual requests for changes, which are known as modification requests, or MRs. Whereas a delta is intended to keep track of lines of code that are changed, an MR is intended to be a change made for a single purpose. Each MR may have many deltas associated with it. The project under consideration used the Sablime system for problem tracking and an internal system based on the Source Code Control System [22] for most of the version control. It is possible to trace all software modification to an MR (or several MRs). The modifications are typically made for one of the following reasons.

1. Repairing previous changes that caused a failure during testing or in the field.
2. Introducing new features to the existing system.
3. Restructuring the code to make it easier to understand and maintain. (An activity more common in heavily modified code, such as in legacy systems.)

An MR is raised against one release (where the defect is first discovered), but it can be delivered to several releases where the defect may manifest itself. Each release often needs distinct changes because the underlying code base often differs.

MRs include, among other things, the MR reporter, the release in which the MR was discovered, and the date the MR was reported. Attributes include the software load where the code was submitted, the resolver, resolution date, and resolution status for each release to which the MR was submitted. We used the CMS and VCS to obtain the new feature and enhancement changes in each release as well as changes in response to failures detected after the software was deployed (field problems).

The text abstracts of the MRs related to a customer issue include information about the customer(s) involved, software releases they are running, and service tickets, filed by customers, that triggered the development MR(s). MRs represent defects (duplicate MRs are identified and closed and are not considered in the analysis), and service and Tier IV tickets represent failures. Therefore, there is a many-to-many relationship between the two. When we look at customer facing measures we do not double count a failure caused by multiple MRs (defects) and when we look at the development process we do not double count defects (MRs) that caused multiple failures.

To calculate the size of a release we count all MRs that submitted code for the release. We use the release and load

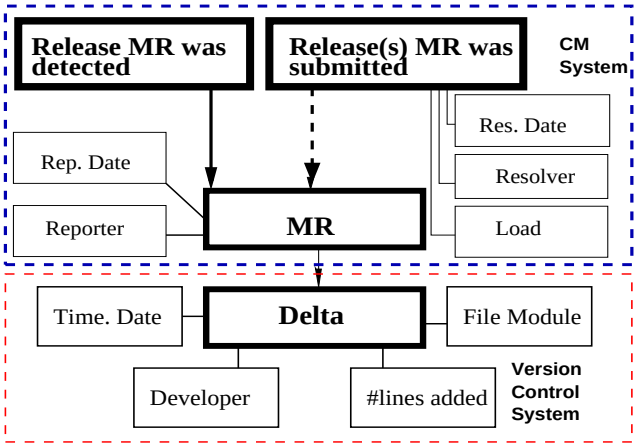

Figure 1: Hierarchy of changes and associated data sources. Boxes with dashed lines define data sources (VCS and CMS), boxes with thick lines define changes, and boxes with thin lines define properties of changes. The arrows define an "is a part of" relationship among changes, e.g., each MR is a part of a release. Dashed arrows indicate many-to-many relationships. Attributes connected to "Release(s) MR was Submitted" are specific to the MR-Release pair.

(build number) to identify the target release. The date on which a release becomes available to customers is known as its General Availability, or GA, date. MRs submitted to loads that are built after the GA date of that release are considered to be patch MRs. Only MRs (and submitted code) for loads that precede GA are included in the release size measures.

To count field faults associated with a release we count each MR only once. We count the MR against the release in which the MR was detected. Some releases were deployed several years ago, while others were deployed fairly recently. To make the comparisons between the releases we restrict the number of defects to the MRs found within ten months of the GA date (based on the latest release we considered).

3.3 Advantages and pitfalls

There are a number of advantages to utilizing existing project support systems, such as the CMS and VCS, when modeling software development and support. Probably the most obvious advantage is that the data collection is non-intrusive. However that does not reduce the need for in-depth understanding of a project's development process and, in particular, of how the support systems are used.

We benefit from a long history of past projects whose data has been captured in project support systems, enabling historic comparisons, calibration, and immediate diagnosis in emergency situations. In some cases, additional data collection may be needed to facilitate modeling, as was in our case when we wanted to obtain the dates when systems were deployed in the past.

The information obtained from the support systems is often fine grained, at the MR/delta/trouble ticket/customer installation level. However, links to aggregate attributes,

such as features and releases, is often tenuous. Furthermore, there may be challenges when cross-linking project support systems in different domains, such as the equipment database and the CMS.

The information tends to be complete, as every action involving development or support is recorded. However the information about what the action pertains to may be non-trivial to infer and some of the data entries, especially those not essential for the domain of activity, tend to be inconsistently or rarely supplied.

The data are uniform over time as the project support systems are rarely changed since they tend to be business-critical and, therefore, very difficult to change without major disruptions. That does not, however, imply that the process was constant over the entire period one may need to analyze.

Even fairly small projects contain large volumes of information in the project support systems making it possible to detect even small effects statistically. This, however, depends on the extractability of the relevant quantities.

The development tools, such as, configuration management and version control systems are used as a standard part of the project, so the software project is unaffected by experimenter intrusion. We should note that this is no longer true when such data are used widely in organizational measurement. Organizational measurement initiatives may impose data collection requirements that the development organizations might not otherwise use and modify their behavior in order to manage the measures tracked by these initiatives.

The largest single obstacle for using the project systems for analysis is the necessity to understand the underlying process and the way the systems are used. This requires validation of the values in fields used by the developers and support technicians in order to assess the quality and usability of the attribute. Common and serious issues involve missing and, especially, default values that may render an attribute unusable. Any fields that do not have a direct role in the activities performed using the project system are highly suspect and, often provide little value in the analysis. As the systems tend to be highly focused to track issues or versions, extracting reliable data needed for analysis may pose a challenge.

4. QUANTIFYING CUSTOMER PERCEIVED QUALITY

There are a variety of ways to measure customer perceived quality, for example surveys and failure rates. We focus on customer-observed reliability, often characterized as MTBF [20] because we found the customer surveys to be inadequate.

4.1 Interval Quality

Previous work indicated the critical importance of the size of a software release [10, 16], the proximity to the release date [17], and the proximity to the installation date [17] as the primary drivers of software issues observed by a customer. To ensure that our measures are useful in practice, we work with the quality experts in the development teams to arrive at a software focused measure of customer satisfaction that can be used in managing a project. In addition to reflecting the primary drivers, such a measure has to be easy to calculate, easily understandable by a variety of people in different roles, and available early enough to allow time for corrective action.

To satisfy our goals we chose a measure that calculates empirically the probability of a customer observing a software issue within a short period after software installation. Periods of 1, 3, and 6 months were chosen to allow faster calculation of the measure (1 month) and a more robust estimate of quality (6 months). For the one-month-metric we have to wait at least one month after GA to start observing the customers who had a full month of experience running the system. One month is long enough to get an early indication of quality and short enough so that a development team can act quickly enough to take corrective actions if the perceived quality is problematic. The six-month-metric requires at least six months wait at which time it may be too late to take corrective action. Therefore, while it is more accurate than the one-month-metric, it can provide value only in a historic context, limiting its practical applications.

To make the measure simple to understand we had to exclude an important factor from consideration, i.e., the proximity to the general availability (GA) date. In [17] we found that the first customers to receive a new release had a much higher probability of reporting a software-related issue than later customers. The most probable underlying causes included the lack of installation and configuration skills (a misconfigured system is often more likely to reveal a latent fault) early in the deployment and delivering patched systems for later customers. This early-customer trend implies that the quality of the latest release is estimated conservatively at first, because at that time only customers who had deployed immediately after GA are represented by the metric. This drawback was not deemed to be serious, in fact, it could provide impetus for more aggressive action.

Because these metrics are empirically calculated based on intervals after product release we call them collectively the interval quality, or IQ.

The probability that a customer observes a failure within one month of installation is estimated as the fraction of customers that had an issue within one month of installation. The population of all customers who had the system installed or upgraded with the particular release between one and seven months after the GA date is used. We also exclude systems that had the release for less than one month. We chose the seven month period to be able to calculate the 3-month probability for the latest release we consider. Another reason was that a new release often appears within seven months. Finally, the failure rate drops as we move further from the GA date, making it less likely that an unacceptably high failure rate would be observed.

When we do the calculation for the three- and six- month intervals, the population of systems declines as some systems are upgraded to newer releases during this period and, therefore, do not stay on the same release long enough to be included in the sample. Given the very low probability of the failure we need very large sample sizes to get reasonably accurate estimates. Therefore we estimate three probabilities separately:

1. $\hat{p}_1$ — the fraction of systems to have a failure in the first month of usage as described above;

2. $\hat{p}_{2-3}$ — the fraction of systems to have a failure in months two and three of usage but no failure in the first month;

3. $\hat{p}_{4-6}$ — the fraction of systems to have a failure in months four through six of usage but no failure in the first three months.

The estimate of the probability of the failure in the first three months is $\hat{p}_1 + \hat{p}_{2-3}$ and the estimate of the probability of the failure in the first six months is the sum of all three estimates.

In addition to providing a more accurate estimate this composite estimation ensures consistency so that the estimate for one month is always less than the estimate for three months and the estimate for three months is always less than the estimate for six months.

We also compare the estimates to a previous release to determine if there are significant differences.

4.2 Other measures

We have compared this measure to a number of other measures that have been used in software projects. We considered the defect density that was successfully used in, for example, [9], and the MTBF. Finally, we also considered the change in business and management priorities over the considered time period.

4.2.1 Post GA defects

We expected the number of post-GA defects to correlate with the release size and customer perceived quality, reflecting the fact that major releases tend to have more defects and tend to be perceived as less reliable by customers. We have analyzed defects found up to 10 months after the GA date to make the numbers comparable between older releases that have been in the field for many years and the most recent release. As a lower bound we calculated the number of post-GA MRs that could be traced to individual customer(s), while an upper bound was the total number of field MRs raised against a release.

4.2.2 Post GA defect density

Dividing the number of post-GA (field) problems by the number of pre-GA code submission MRs or by the non-commentary source lines of code (NCSL) that were added, modified, or deleted in a release should provide a characteristic of how good the development process was. We expected the reductions in this defect density to be reflected in the improvements of customer perceived quality.

4.2.3 Number of Tier IV tickets and MR related tickets

We expected the number of Tier IV tickets to reflect customer perceived quality. We considered only tickets for systems installed or upgraded within seven months of the GA date and tickets created within ten months of the GA date to preserve comparability between old and new releases and to the field MR counts that represent the same ten month periods.

4.2.4 NCSL, pre-GA MRs and Runtime

NCSL, pre-GA MRs and Runtime are size measures needed to calculate the defect density and MTBF. The pre-GA MRs were obtained from the Sablime system [14]. MRs that did not contribute code to the release in question or MRs that were created after the GA date were excluded. The NCSL added and changed were collected for every MR as a part of the load process. We have included NCSL for all the pre-GA MRs related to a release.

Runtime for a system was calculated as the interval between the installation and 10 months after GA or the time of upgrade to the next release, whichever was less. Runtimes were added over all the systems installed in the 7 month period following the GA. These fixed periods were used in order to make comparisons between recent and older releases.

4.2.5 MTBF

The MTBF can be estimated by dividing the total number of faults by the total running time of the deployed systems. We expected this measure to follow IQ closely. Here we consider software specific faults such as the Tier IV tickets and the trouble tickets that result in an MR. Only systems deployed within seven months of GA and tickets from such systems created within ten months of GA were considered for the reasons described above.

5. COMPARISON

In order to evaluate which other measures may be most reflective of the IQ we present and compare them in the following subsections. We first provide the measures validated as described below in order to encourage alternative analyses or interpretations by other researchers. Any interested reader can easily perform additional analyses using the provided measures. Finally, we hope that the availability of such information will encourage advances in software engineering and will provide incentive to evaluate customer perceived product quality based on quantitative information. The latter would encourage customers to put pressure on other software organizations to start collecting and publishing similar data. Such an outcome would be likely to advance software engineering practices and to bring benefits to customers and software providers.

Obtaining and validating measures in Table 1 involved a substantial multi-year effort to learn about the various processes and the supporting systems from which the data were retrieved. It is substantially easier to obtain and validate a single type of data, such as software defect density in development, because quality engineers and expert developers exist for a project who may already track and validate some or all of the relevant quantities. Unfortunately, it is much less likely to find someone who has an in-depth understanding of all the parts involved when considering the entire process from development to sales and then support. This difficulty of integrating different processes and data collected for the internal process needs may partly explain the dearth of empirical studies linking development process improvements to improved customer perceptions. Indeed, the results presented here may be unique at this time, and we hope that it will motivate more studies in this challenging area.

In addition to development and service organizations, the information from ordering systems had to be integrated to obtain the number and properties of the installed base. We had to warehouse weekly snapshots of the installed base over four years to be able to reconstruct accurate installation or upgrade dates in the past. The operational systems could be easily enhanced to query such information directly, but the lack of users with such analytic needs has left such functionality missing.

The measures in Table 1 originate from development, or-

dering, and support domains. The development domain contains the size of the releases in thousands of NCSL (KNCSL) and in the number of pre-GA MRs (pre-GA) submitted to each release. The problems are represented by counts of field MRs (Field), and field MRs traceable to the reporting customer (Field-C).

Ordering information is used to determine the date of installation or upgrade, and the properties of the system. It allows calculation of the installed base (NSys) for each release over the periods of interest and calculation of the total runtime (in years) of such systems (Runtime).

The service domain includes counts of MR-related tickets (MR tickets), counts of Tier IV tickets (Tier IV), and our quality measures for the probability of Tier IV and MR events (multiplied by 1000 for better readability).

There were two major releases "r1.1" and "r2.0". The rest were minor releases. We should note that "r1.1" was not the first release of software - there were many releases before that.

5.1 Defect density

We start by investigating how well the traditional defect density measure is correlated with the perceived quality measure. It makes sense that decreasing defect density is good practice and that it should lead to improvements in customer perceived quality. The reality, as it turns out, is somewhat complicated by the fact that the customer perceived quality may be affected by a variety of other practices.

We prefer to use at least two operationalizations of defect density and, in this case we use two scalings, one by NCSL and one by the number of pre-GA MRs. Figure 2 indicates the lack of correlation with IQ. In fact the correlation (we use Spearman correlation as the distributions are not normal) is negative, -0.7, and is significantly less than zero (p-value of .07) in the case of MR normalized density with both quality measures. The signal in the data must be very strong for such a small sample, i.e., six data points, to have statistical significance. The two operationalizations of defect density had a correlation of .83 and the correlation was significantly greater than 0 (p-value of .03).

In fact, we see that the defect density is lowest for major releases and highest for minor releases, while the quality measure shows the opposite behavior. From the numeric perspective the defect density calculates defects per line of code or per change and in larger releases the denominator creates the trend as the number of defects is relatively constant (see Table 1). In the quality measure the denominator represents the number of customers and is larger in minor releases that are deployed more broadly.

From the release deployment perspective it is important to resolve defects on a timely basis. Such resolution requires significant staffing resources that can be best utilized if the work-flow is relatively constant. Therefore, deploying more fault prone releases gradually and the most robust minor releases more rapidly results in a relatively constant inflow of issues that can be handled in the most efficient manner. This, however, disassociates the traditional defect density measure from the probability that a customer will experience a software issue.

We do not imply that the lack of positive correlation somehow reduces the need to improve defect density, as improvements in defect density could allow more rapid deployment or lower staffing for support and Tier IV. However, it is important to be aware that the deployment strategy may more than compensate for the variations in the defect density and, therefore, the improvements in the software process may not be always reflected in the customer perceived quality.

5.2 MTBF

Mean time between failures is a common hardware reliability measure. It may not be used as often for software partly because it is not easy to separate software and hardware issues and partly because it is quite difficult to estimate for software. Furthermore, hardware (especially mechanical hardware) tends to wear out, while software does not have an analogous property. Software failures are more difficult to count (unlike the hardware faults that require replacement parts that leave traces in ordering and support systems). It is often harder to get precise information on the deployed software base, because, unlike hardware, software can often be upgraded without producing traces in the dispatch or ordering systems.

We use calendar time rather than run-time because we think it more closely accords with customer perceptions of quality and there is no way we can observe actual run time. In fact, the considered systems are intended to run continuously in the field, except when there is a failure or an upgrade.

The MTBF is impacted by the proximity of the installation date to GA, by the time elapsed after installation, and the utilization of the system as was shown by models in [17]. Therefore, to make release numbers comparable we restrict the calculation to customers installing within 7 months of GA to make sure we have sufficient data for the latest analyzed release. We also restrict observation to faults observed in these installations within 10 months of GA to ensure that all considered systems had sufficient runtime.

It is important to note that MTBF as calculated does not represent MTBF of a long-running system since the probability of observing failure drops dramatically with time elapsed from the installation and from the GA dates. However, it is useful to compare the behavior of releases close to installation and GA dates — times when it is most likely for a customer to encounter an issue.

Availability is a related measure representing the percent of time the system is up and running (available for operations) and requires the knowledge of an average outage duration in addition to the MTBF. It is very difficult to determine an average outage interval precisely, therefore we present availability numbers for a range of such intervals. The outage durations vary widely from a few seconds, where outages are repaired in software, to hours and, possibly, days when a dispatch is needed to replace failed hardware or a workaround or patch has to be developed for a complex software issue.

The considered systems had to satisfy various availability requirements. Most had three nines (available 99.9% of the time) availability requirements, but many have requirements of five nines. We expect the lower availability requirements systems to dominate our sample as there are far more of them. Given realistic estimates of outage times it appears (see Table 2) that most of the systems far exceed the availability requirements, at least from the software perspective. For example, the table shows that for the empirically calculated MTBF of release 1.1, which was 297921 hours, achiev-

	Releases	r1.1	r1.2	r1.3	r2.0	r2.1	r2.2
Development Domain	KNCSL	376	64	154	235	188	60
	pre-GA	2971	1190	1155	2189	1408	594
	Field	112	99	105	96	100	92
	Field-C	61	68	68	51	60	54
Ordering Domain	NSys	4575	6398	10943	8417	16168	14073
	Runtime	2005	3780	5908	3903	8477	3594
Service Domain	MR tickets	59	63	78	78	111	20
	Tier IV	303	307	382	293	515	174
	T4Quality1M	19	13	10	11	7	7
	MrQuality1M	9	5	4	4	2	2
	T4Quality3M	37	28	21	20	16	17
	MrQuality3M	16	12	7	8	5	3
	T4Quality6M	52	41	30	26	26	
	MrQuality6M	23	16	10	9	8	

Table 1: Primary measures

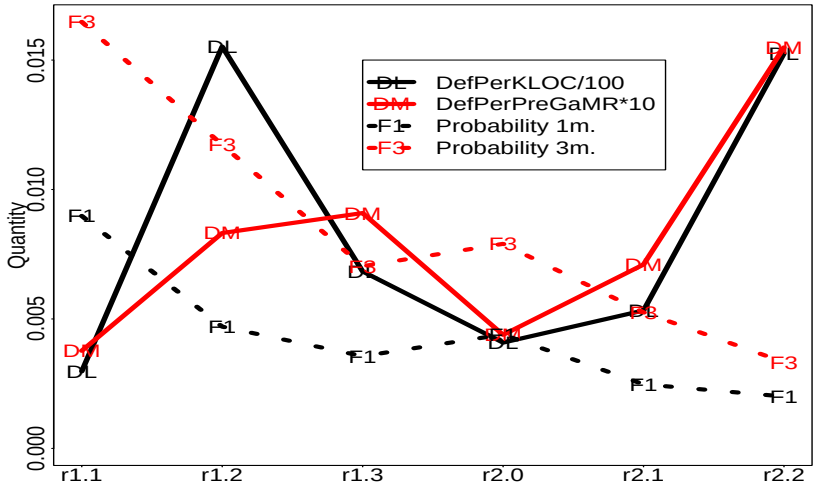

Figure 2: The trend of the defect density and quality measures over releases.

ing .999 availability requires an average outage time of 298 hours. Since 298 hours is clearly much more than any sensible estimate of the average outage time, even release 1.1 easily meets 3 9s availability. It is worth noting that a significant portion of customer issues that result in a software change do not cause service outages, therefore the presented numbers only provide a lower bound on MTBF and availability.

	Type	MTBF	.999	.9999	.99999
r1.1	Major	297921	298	30	3
r1.2	Minor	525995	526	53	5
r1.3	Minor	663948	664	66	7
r2.0	Major	438672	439	44	4
r2.1	Minor	669480	669	67	7
r2.2	Minor	1575329	1575	158	16

Table 2: MTBF and the required average outage times for three availability levels. Intervals are measured in calendar hours.

Although MTBF behaves similarly to the failure probability measure, it mixes recently and long-ago installed systems. This prevents observing the drop in failure probability with time expired after the installation date. It also requires even more data (runtime) than the measures of failure probability.

5.2.1 *Change in business priorities*

The first product release we considered in this work was a release that was motivated by time-to-market needs that were emphasized by the fact that Avaya was a new company, although a spin-off of Lucent and therefore initially staffed with many experienced software developers, and needed radically new products in the market to survive.

The impact on the product development organization was profound, especially since it was not used to a rapidly-changing market situation. The resulting organizational changes were documented in an in-depth goal-oriented assessment [27] of the project six months before completion.

Figure 3 presents the quality measures over a sequence of releases. Significantly different estimates from the previous release are indicated with a star(s) at the top of the bar. One, two, and three stars correspond to p-values of .1,

.05, and .01. We can see quality improve; a number of the decreases are significant.

Over time business priorities evolve and tend to change focus. While it is difficult to quantify the "focus", let alone the change in focus, we feel that there are cyclical trends in software-based industries, as in many other industries, cycling among quality, cost, and time-to-market. The primary reason may be the fact that these qualities have to be traded off in order to improve one of them. When the emphasis is on time-to-market, quality may suffer. As we mentioned, in the beginning of the considered period there were real and perceived reasons for quickly developing IP based solutions, while in the subsequent years the downturn of the industry and maturing of the technology caused the focus to move towards reliability and availability. We believe that one reason for the trend in improved IQ in Figure 3 is a shift in business priority towards quality after an initial emphasis on time-to-market. The approximate interval over which we have seen a number of indications (organizational initiatives, management directives, and activities of the development organization) for the shift in priority from time-to-market to customer perceived quality occurred during releases "r1.3' and "r2.0" and is depicted in Figure 3. In particular, we believe that the IQ of the major release "r2.0" would be higher than IQ of the previous minor release "r1.3" if not for the shift in focus.

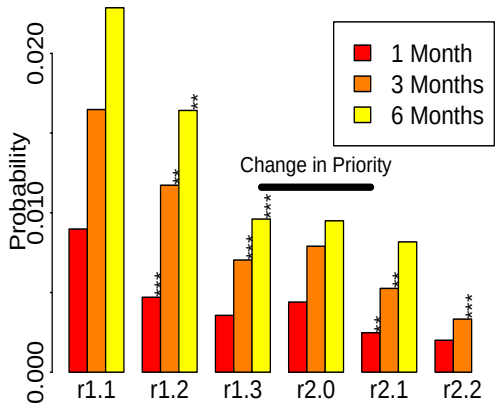

Figure 3: The trend of the IQ over releases.

6. VALIDATION

Does the proposed measure of customer perceived quality accurately reflect customer perceived quality? Our measure reflects some aspects of installability (a significant portion of issues involve installation/configuration, especially for early adopters of major releases, see [17]) and aspects of maintainability (the remaining issues observed over the six month period). It clearly does not capture feature-richness, often a strong customer satisfier. In fact, it may be that the overall quality (if it could be quantified) may stay constant as the releases with the most radical new features (presumably valued by customers) are balanced by the larger likelihood to experience problems.

Another aspect of quality represents the time it takes to provide a fix. In, for example, [6], the amount of time to resolve an issue is closely related to customer satisfaction measured through a survey. While IQ does not capture this aspect directly, it appears to be important from the business perspective. In particular, the steady stream of issues provides a constant work-flow for the services and Tier IV. The lack of peaks in problem inflow intensity (which can be made possible by using an appropriate deployment strategy) provides a steady stream of work. The organization can then anticipate and assign sufficient staffing to make sure that the work items are handled on a timely basis.

At this point in time we believe our approach to be the most practical for the considered types of systems and the quality aspects that summarize faults and installation issues. Our belief is based on our experiences in developing the measure and on the amount and quality of data that may be commonly available. The use of IQ in the production environment suggests its value for internal quality management.

A direct approach to validate the customer satisfaction measure with customer satisfaction surveys, see, for example, [3], turned out to be problematic in our case. First, for the analyzed back-office-systems that have massive numbers of end users, it is not clear which users to target. In fact, the recorded outage information is likely to provide more accurate results than a sample of users. The existing surveys are not uniform over time and tend to focus on IT managers who make investment decisions that strongly depend on factors that may only tangentially depend on the observed problems.

In addition to understanding how IQ reflects customer perceived quality we spent a lot of effort to make sure that the operational quantities obtained reflect the reality. We have many years of experience extracting and cleaning data from software project support systems such as version control and problem tracking systems. While the full description of the methodology is beyond the scope of this submission, we highlight some specific issues related to integrating information from diverse development, customer support, and ordering systems.

We first familiarized ourselves with the variety of ways the systems are used and the types of records kept. As many of the systems (especially in development support) do not keep data in relational databases, methods to recognize, extract, and validate relevant fields were needed. In particular, the information about a customer and the support ticket was embedded within the text of the MR description in various formats that had to be recognized and validated. Below we describe some basic principles we used to obtained desired quantities, including focusing on the requirements for our analysis and on basic validation based on multiple operationalizations, cross-linking systems, and obtaining bounds on desired quantities. We also discuss model-level validation where normalization or subset selection is based on well-known properties of software development and product support.

6.1 Extract according to objectives

After familiarizing ourselves with the project support systems we analyzed the properties of the extracted data to validate the reliability of various attributes and their relevance to the measures of interest. In particular, we found that only a subset of MRs that submit code affect the release content. Code that does not end up in a release can not directly contribute to customer observed issues. Our objective was focused on what could contribute to problems (MRs in-

volved in code submissions to a release) and the problems themselves (field MRs).

It is worth remarking that the notation used to denote a release was different depending on what field is being used to determine the release. For example, the field for "release detected" used different notation than the field indicating the releases where the MR was submitted. The notation has also changed over time. We had to extract and inspect all of the different recorded values to assign them to the corresponding releases. Of course, the notation used to denote releases in service issue tracking systems was distinct from the one used in development systems.

As noted previously, a single MR had to be counted in all releases to which it was submitted because the submission introduced a possibility of a defect. However, we counted an MR only once for the release in which it was detected to make sure we count unique defects. The submissions of the same MR in newer releases were intended to prevent potential failures.

In cases where there was a lack of relevant data (historic deployment information) we had to set up additional automatic data collection facilities and, over the last four years, have collected information on a number of releases sufficient for the analysis.

6.2 Multiple operationalizations and integration

We always try to use multiple operationalizations of the same measure. Mismatches between operationalizations are investigated and may indicate errors in data extraction or processing, changes (or the lack of understanding) of the underlying process, or high sensitivity of the measure itself. We try to design different operationalizations in a way that provides the upper and lower bounds on the quantity in question. For example, the total number of field MRs represents an upper bound on the number of known defects, because some field MRs are enhancements delivered after the GA date. The number of field MRs that can be traced to customers or trouble tickets provides a lower bound, because there may be field MRs where a customer or a trouble ticket are not identified.

This example also illustrates another practice useful when integrating data from different systems. While each system focuses on its main purpose (MRs on software versions, and trouble tickets on customer issues) some attributes such as the trouble ticket number or a customer system ID may appear in MR systems, or the MR number may appear in Tier IV or trouble ticket systems. Such a link often establishes the lower bound on the number of entities that pertain to another domain as the occurrence of these IDs is usually entered in a free form text field and may be missed.

6.3 Basic normalization

We have taken steps, where possible, to adjust the observed quantities so that we are not simply observing the differences that are known from existing models and experience to be caused by different times of exposures or differences in size or complexity. For example, to make sure that longer exposure of the older releases does not bias the results, we have selected only events within ten months of the GA for each release, so that all the compared releases have similar time exposures. Similarly we make the customer populations in the quality metric similar, by ensuring they had identical runtime and were deploying in similar intervals from the GA date (both factors are known to affect the probability of observing defects). The fact that we are looking at the probability of a customer observing a failure adjusts for the number of customers — having no customers would make software perfect because no failures (and hence no post-GA defects) would be found.

When investigating the defects we adjust for the release size, because given the same deployment rate the larger releases would result in more discovered faults. That leads to investigation of the defect density, not of the number of defects.

6.4 Other considerations

Even over the short period of four years presented here, there were significant changes in the systems used. For example a different system was introduced to track Tier IV tickets that we had to integrate into our analysis.

We have heavily utilized in our work the previous experience on quality and other organizational metrics gained in this and other products.

To verify that the approach works more broadly we have applied the technique on a completely different product and obtained similar results.

Despite all the steps we have taken, the observational nature of the study limits us from drawing causal inference.

7. SUMMARY

Software organizations are complex and must plan for or rapidly react to a number of eventualities. Therefore, observational studies of such organizations have to look at many different aspects of their behavior to assess what changes or improvements affect customer perceived product quality. Only an extensive cost-benefit analysis can provide rational answers of the relative effort that needs to be spent in requirements identification and specification, design, development, verification, and deployment. The optimal solution is likely to differ depending on the types of products, the customer segments, and the reliability and availability requirements of the particular business segment.

We have developed and evaluated IQ, a practical measure of customer perceived quality based on the probability that a customer will observe a software problem calculated from information that could be obtained from software problem tracking and customer support systems. IQ may therefore be applicable in a wide variety of projects. IQ is used in Avaya to determine if the quality of the releases is improving over time and can be used to trigger quality improvement efforts ranging from slowing deployment to increasing staffing in order to resolve customer issues on a timely basis.

In our study we have observed a lack of association between the commonly used software quality measures such as defect density and the improvement in customer perceived software quality. We found that organizational change in focus from time-to-market to customer quality has been reflected in several measures of customer perceived software quality. We also noticed that the inflow of work from customer problems was quite uniform over time, suggesting that the defect density is balanced with deployment strategies to control the inflow of work and to ensure that there are sufficient resources to handle this inflow efficiently and in a timely manner.

Furthermore, we considered defect density and MTBF

along with a number of other measures and evaluated them in comparison to IQ. We hope that our work will provide incentive to evaluate customer perceived product quality based on quantitative information. We also hope that the publication of such information will encourage customers to request their software providers to start collecting and publishing similar information.

We believe that the presentation of software development, product support, and software reliability measures is unique, and may encourage more research in this important area and may trigger similar publication from other products leading to advances in software engineering practices and bringing benefits to customers and software providers.

Acknowledgments

We thank E. Moritz for her expertise, motivation, and contributions in developing the quality measure.

8. REFERENCES

[1] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as qualityindicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[2] V. Basili and D. Weiss. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Trans. on Software Engineering*, February 1985.

[3] M. Buckley and R. Chillarege. Discovering relationships between service and customer satisfaction. *Proceedings of the International Conference on Software Maintenance*, pages 192 – 201, 1995.

[4] D. A. Christenson and S. T. Huang. Estimating the fault content of software using the fix-on-fix model. *Bell Labs Technical Journal*, 1(1):130–137, Summer 1996.

[5] S. Chulani. Coqualmo (constructive quality model) a software defect density prediction model. *Project Control for Software Quality*, 1999.

[6] S. Chulani, P.Santhanam, D. Moore, and G. Davidson. Deriving a software quality view from customer satisfaction and service data. *European Conference on Metrics and Measurement*, 2001.

[7] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. VanderWiel. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, Melbourne, Australia, May 1992.

[8] E. N. Fenton and P. S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, 20 Park Plaza, Boston, second edition edition, 1997.

[9] R. Grady and D. Caswell. *Software Metrics: establishing a company wide program*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[11] M. H. Halstead. *Elements of Software Science*. Elsevier – North Holland, 1979.

[12] J. Jelinski and P. B. Moranda. Software reliability research. In W. Freiberger, editor, *Probabilistic Models for Software*, pages 485–502. Academic Press, 1972.

[13] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, Dec. 1976.

[14] A. K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[15] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April–June 2000.

[16] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *2003 International Conference on Software Engineering*, pages 274–284, Portland, Oregon, May 3-10 2003. ACM Press.

[17] A. Mockus, P. Zhang, and P. Li. Drivers for customer perceived software quality. In *ICSE 2005*, pages 225–233, St Louis, Missouri, May 2005. ACM Press.

[18] S. N. Mohanty. Models and measurements for quality assessment of software. *ACM Computing Surveys*, 11(3):251–275, September 1979.

[19] J. C. Munson and T. M. Khoshgoftaar. Regression modelling of software quality: Empirical investigation. *Information and Software Technology*, pages 106–114, 1990.

[20] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company, 1987. ISBN: 0-07-044093-X.

[21] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. on Software Engineering*, 22(12):886–894, December 1996.

[22] M. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, 1(4):364–370, 1975.

[23] G. J. Schick and R. W. Wolverton. An analysis of competing software reliability models. *IEEE Trans. on Software Engineering*, SE-4(2):104–120, March 1978.

[24] N. F. Schneidewind and H.-M. Hoffman. An experiment in software error data collection and analysis. *IEEE Trans. on Software Engineering*, SE-5(3):276–286, May 1979.

[25] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software–an empirical study. *IEEE Trans. on Software Engineering*, SE-11(4):317–324, April 1985.

[26] D. Weiss. Evaluating software development by error analysis: The data from the architecture research facility. *J. Systems and Software*, 1:57–70, 1979.

[27] D. Weiss, D. Bennett, J. Payseur, P. Zhang, and P. Tendick. Goal-oriented software assessment. In *Proceedings of the 24th International Conference on Software Engineering*, pages 221 – 231, Orlando, Florida, 2002. ACM Press. ISBN: 1-58113-472-X.

[28] T. J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Trans. on Software Engineering*, 14(9):1261–1270, September 1988.