Modeling and software development quality

by S. H. Kan

This paper summarizes the models used by a large software development organization for estimating software reliability and managing software development quality. The role of modeling in software quality improvement is illustrated. Implementation of the models, reliability of the estimates, predictive validity, and the nature of data are discussed.

An emerging trend in the software industry is to use scientific methods to achieve precision in managing software projects. In progressing toward this goal, the role of statistical and engineering modeling is pivotal. Modeling is a systematic way to describe the complex reality. The knowledge we gain from modeling can help us develop quality strategies scientifically and manage quality accordingly and appropriately.

This paper briefly describes the models used for estimating software reliability and managing software development quality at the IBM Rochester Programming Laboratory. Examples are based on data collected during the development of the IBM Application System/400® (AS/400®) software system. Issues of implementation, reliability, predictive validity, and the data are discussed where applicable.

Reliability models

Reliability models are used to estimate the defect rate that is latent in a system when the system is shipped. Such an estimate is important for two reasons: 1) it is an objective statement of the code quality of the software system, and 2) it may be used for resource planning in order to ensure continual programming and service support.

A number of software reliability models exist, each having its own associated assumptions, limitations, and applicability (see Reference 1). After examining the data and considering relevant issues, we decided to use the Rayleigh model and the exponential model and its variants. Both models belong to the family of the Weibull distribution. The Weibull distribution has been used for decades for reliability analysis in various fields of engineering, ranging from the fatigue life of deep-groove ball bearings to electron tube failures and the overflow incidence of rivers. It is one of the three known extreme-value distributions (see Reference 2), and one of its marked characteristics is that the tail of its probability density function approaches zero asymptotically, but never reaches it. Its cumulative distribution function (*CDF*) and probability density function (*PDF*) are:

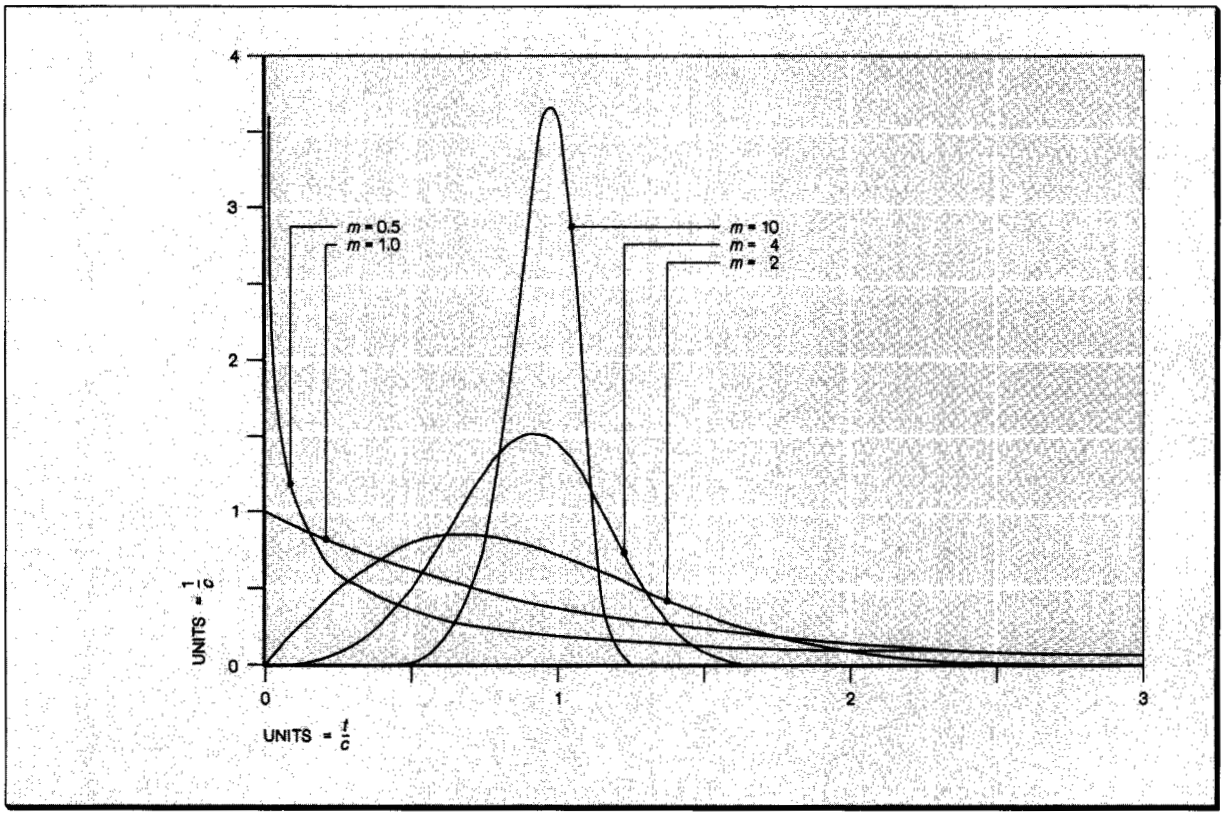
$$CDF\text{: } F(t) = 1 - e^{-\left(\frac{t}{c}\right)^m}$$

$$PDF\text{: } f(t) = \frac{m}{t} \cdot \left(\frac{t}{c}\right)^m \cdot e^{-\left(\frac{t}{c}\right)^m}$$

where m is the shape parameter, c is the scale parameter, and t is time.

Figure 1 shows several Weibull probability density curves with varying values of the shape parameter, m.

Figure 1 Weibull probability density function



The Rayleigh model. The Rayleigh model is a special case of the Weibull distribution when $m = 2$. Its *CDF* and *PDF* are:

$$CDF\text{: } F(t) = 1 - e^{-\left(\frac{t}{c}\right)^2}$$

$$PDF\text{: } f(t) = \frac{2}{t} \cdot \left(\frac{t}{c}\right)^2 \cdot e^{-\left(\frac{t}{c}\right)^2}$$

The Rayleigh *PDF* first increases to a peak and then decreases at a decelerating rate. This is illustrated in Figure 1 for $m = 2$. The c parameter is a function of tm, the time at which the curve reaches its peak. By taking the derivative of $f(t)$ with respect to t, setting it to zero, and solving the equation, tm can be obtained.

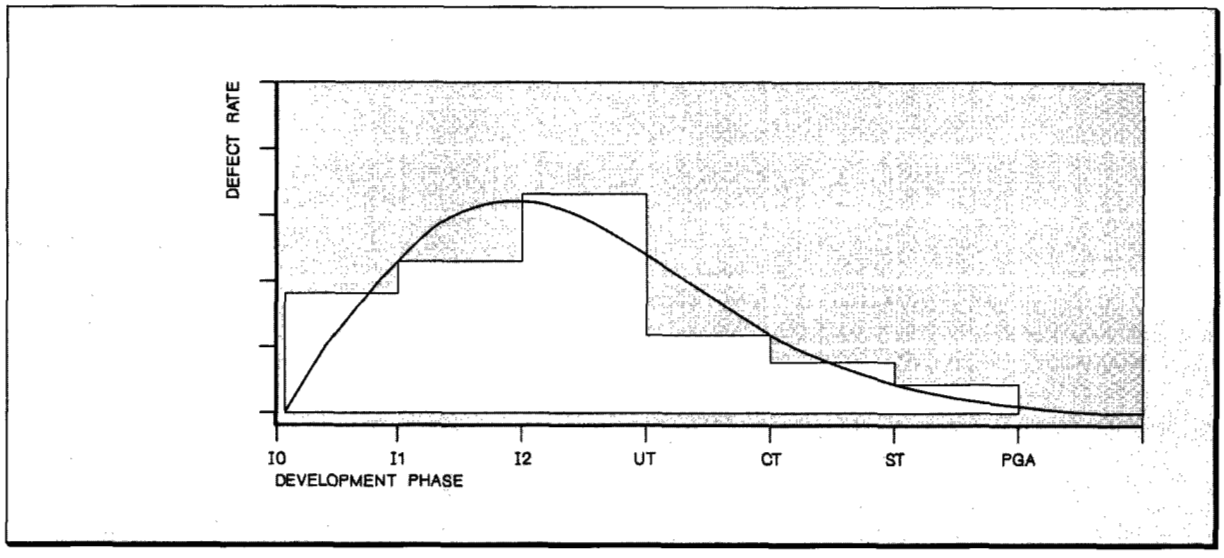
$$tm = \frac{c}{\sqrt{2}}$$

After tm is estimated, the shape of the entire curve can be determined. The area below the curve up to tm is 39.35 percent of the total area.

It has been empirically well-established that software projects follow a life-cycle pattern described by the Rayleigh density curve (see Reference 3). The basic assumption for using the Rayleigh model is that if more defects are discovered and removed in the earlier development phases, fewer will remain in later stages, which results in better quality of the system. As shown in Figure 2, we modeled the defect-removal pattern of the six development steps: high-level design (I0), low-level design (I1), coding (I2), unit test (UT), component test (CT), and system test (ST). The defect rate of the last phase in the figure, the post-general-availability phase (PGA), is the target of our estimate.

The exponential model. The exponential model is yet another special case of the Weibull family,

Figure 2 Rayleigh model



with the shape parameter m equal to 1. It is best used for statistical processes that decline monotonically to an asymptote. Its *CDF* and *PDF* are:

$$CDF: F(t) = 1 - e^{-\left(\frac{t}{c}\right)}$$

$$= 1 - e^{-\lambda t}$$

$$PDF: f(t) = \frac{1}{c} \cdot e^{-\left(\frac{t}{c}\right)}$$

$$= \lambda \cdot e^{-\lambda t}$$

where c is the scale parameter, t is time, and $\lambda = 1/c$.

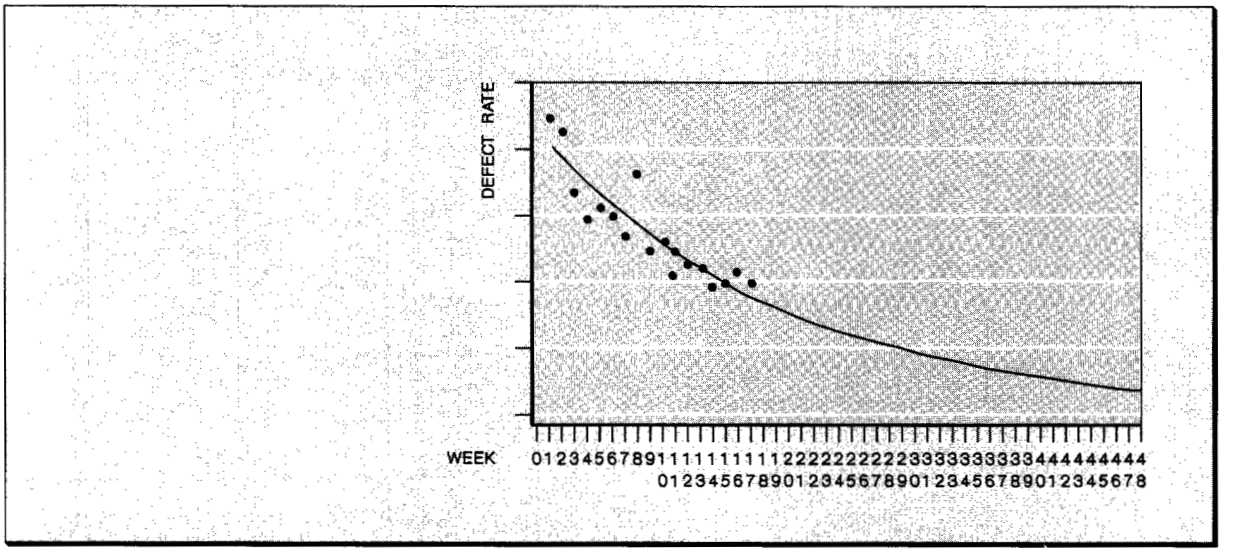
The exponential model is one of the better known software reliability models. The Goel–Okumoto Nonhomogeneous Poisson Process Model (NPPM) is based on the exponential distribution (see Reference 4). Misra (see Reference 5) used the exponential model to estimate the defect-arrival rates for the Shuttle Ground System software of the National Aeronautics and Space Administration (NASA). The software provided the flight controllers at the Johnson Space Center with processing support to exercise command and control over flight operations. Actual data from a 200-hour flight mission indicated that the model worked very well.

As Figure 3 shows, we modeled the weekly defect-arrival data since the start of the system test, which started when the development work was virtually complete. The system test uses customer interfaces, tests external requirements, and simulates end-user application environments. The pattern of defect arrivals during this stage, therefore, should be indicative of the latent-defect rate.

In addition to the standard exponential model, we also used two variant models that were proposed by Ohba for software reliability analysis: the delayed S model and the inflection S model (see Reference 6). The exponential model has no inflection points, the delayed S model has a slight inflection at the beginning of the curve, and the inflection S model has a substantial inflection. The exponential model assumes that the peak of defect arrival is at the beginning of the system test phase, and continues to decline thereafter. The delayed S model assumes a slightly delayed peak, and the inflection S model assumes a later and sharper peak. The three models, as well as others similar, are referred to as reliability growth models.

Implementation. Our estimations of the model parameters of the Rayleigh and exponential models

Figure 3 Exponential model



are implemented by programs written in the Statistical Analysis System, SAS (RAY for the Rayleigh model and EXPONEN for the exponential model). The programs use the nonlinear regression procedure. From the several methods in nonlinear regression, we chose the DUD method for its simplicity and efficiency (see Reference 7). DUD is a derivative-free algorithm for nonlinear least squares. It competes favorably with even the best derivative-based algorithms when evaluated on a number of standard test problems.

Our SAS programs estimate model parameters and produce a graph of fitted model versus actual data points on a graphic terminal screen using the Graphics Data Display Manager (GDDM). The SAS programs perform a chi square goodness-of-fit test, and derive estimates for the latent-error rate. The probability (p-value) of the chi square test is also calculated. If the test results indicate that the fitted model does not adequately describe the observed data ($p > .05$), a warning statement is issued in the output.

In 1985, the IBM Federal Systems Division at Gaithersburg, Maryland, developed a PC program called the Software Error Estimation Reporter (STEER). The STEER program, now available to other IBM sites, implements a discrete version of the Rayleigh model by matching the input data with a set of 11 stored Rayleigh patterns and a number of user patterns. The stored Rayleigh patterns are expressed in terms of percent distribution of defects for the six development phases: high-level design, low-level design, coding, unit test, integration test, and system test. The matching algorithm involves taking a logarithm transformation of the input data and the stored Rayleigh patterns, calculating the separation index between the input data and each of the stored patterns, and selecting the stored pattern with the lowest separation index as the best-fit pattern.

There are several problems with the STEER approach. First, the matching algorithm deviates from statistical estimation methodology, which derives estimates of model parameters directly from the input data points based on theoretically proven procedures. Second, it always produces a best-match pattern even when none of the stored patterns is statistically adequate in describing the input data. There is no mention of how small the separation index should be in order to indicate a good fit. Third, the stored Rayleigh patterns are far apart; specifically, they range from 1.00 to 3.00 in terms of tm, with a tm increment of 0.25, which is considerably large. This means they are not sufficiently sensitive for estimating the latent-error rate, which is usually a very small number.

There are, however, ways to circumvent the last two problems. First, use the separation index conservatively and be skeptical of the results if the index exceeds 1. Second, use the program iteratively: after selecting the best-match pattern (for instance, the one with $tm = 1.75$), calculate a series of slightly different Rayleigh patterns that center at the best-match pattern (for instance, patterns ranging from $tm = 1.50$ to $tm = 2.00$, with an increment of 0.05 or 0.01) and use them as user patterns to match against the input data again. The outcome will be a better "best match." With this approach, STEER could be a good reference tool when mainframe statistical tools are not available.

The Software Error Tracking Tool (SETT) developed by Falcetano and Caruso (see Reference 8) provides an implementation for the three reliability growth models discussed earlier.

Reliability and predictive validity. Reliability refers to the degree of change in the model output due to chance fluctuations in the input data. In the present case, the question is how accurate are the software reliability estimates. In specific statistical terms, reliability relates closely to the confidence interval of the estimate: the narrower the confidence interval, the more reliable the estimate, and vice versa.

We purposely did not form confidence intervals on our estimates. Given the few available input data points, the chance of having satisfactory confidence intervals is very slim. This is especially true for the Rayleigh model, which is based on only six rates. Our strategy, therefore, is to rely instead on intermodel reliability. Although the confidence interval for each model estimate may not be satisfactory, if the estimates by different models are close to each other, our confidence about the estimates is strengthened. On the other hand, if the estimates from different models lie far apart, our confidence will not be strengthened even if the confidence interval for each single estimate is very small. In our case, the Rayleigh model and the exponential model are based on entirely different data. The final estimates for the two approaches turned out to be consistent, and we are comfortable with the results.

Predictive validity refers simply to the accuracy of model estimates. To establish predictive validity, model estimates and actual outcomes must be compared. When applying the Rayleigh model to study the life cycle of manpower of software projects, Wiener-Ehrlich and associates found that the model underestimated the manloading scores at the tail (see Reference 9). At the IBM Rochester Programming Laboratory, actual defect data after the system is shipped are available for several releases of the IBM System/38. When comparing the Rayleigh estimates with the actual defect rates, we also found that the Rayleigh model consistently underestimates the latent-error rates. Using the Rayleigh estimates directly, therefore, will not be predictively valid. Our strategy is to adjust the Rayleigh output based on the IBM System/38 experience. The adjustment factor is the mean difference between the Rayleigh estimates and the actual defect rates reported. The adjustment is logical, given the similar architectures of IBM AS/400 and IBM System/38 and the similar structural parameters in the development process, including organization, management, and work force.

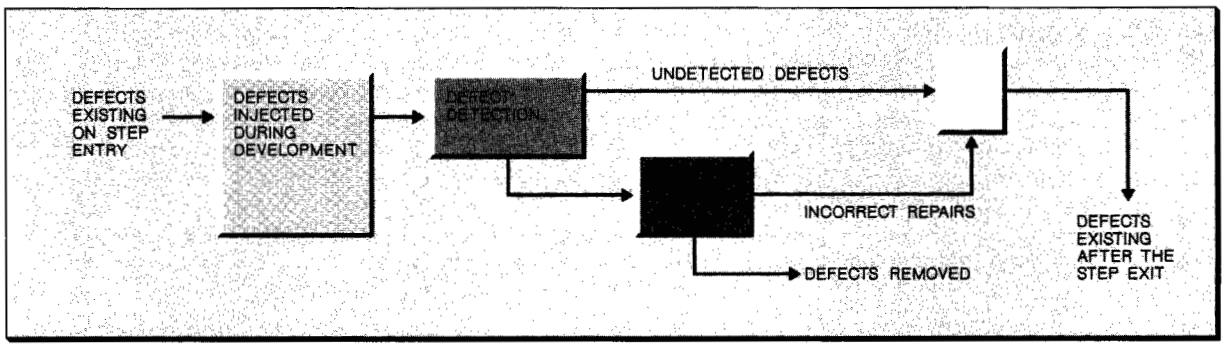
The exponential model is well known in software-reliability analysis. However, most of the model's applications in the literature were for "clean" execution-time data. The defect-arrival data we have are calendar-time based. In other words, the data are less precise. Ohba notes the exponential model does not work well for calendar-time data with a nonhomogeneous time distribution of the testing effort (see Reference 6). In the case of the IBM AS/400, the testing efforts remained consistently high and homogeneous throughout the system-test phase. Furthermore, because the system is very large (7.1 million lines of source code at the first release), the rates tend to be stable even though no execution-time data are available.

The general availability (GA) date of the IBM AS/400 was August 24, 1988. Since then, the actual field defect arrivals have been tracked. Projections based on defect arrivals validated our estimation methodology based on pre-GA data.

Software quality management models

It is important to assess the quality of a software system when development work is complete. It is as important, if not more so, to monitor the quality during development. Software quality management models serve as the measuring tool. Early warning signs or improvement can be detected, and timely management actions can be taken.

Figure 4 Defect injection and removal step



Any software quality management model must cover the early development stages in order to be useful. Models that are based on data collected at the end of the development process permit little time for actions, if needed. The exponential model, which is based on system-test data when development work is virtually complete, is, therefore, not as useful as other models that are discussed in the following sections. However, when sufficient system-test data are available, the exponential model can be used to determine when to end system testing for a specific, predetermined quality index.

The Rayleigh model. A Rayleigh model derived from a previous release of a product or from historical data can be used to track the current pattern of defect removal. If the current pattern is more front-loaded than the model would predict, it is a positive sign; it is a negative sign if the opposite is true. When sufficient data are available before development work is complete, parameters of the model can be re-estimated, and projections of the final quality index can be made. For models based on defect-removal rates of development steps (in our case, six steps), at least two rates are required for parameter estimation. Such early projections would not be as reliable as the final estimate at the end of the development cycle. Nonetheless, they can indicate the direction of the quality of the current release so that timely actions can be taken. For instance, if projections at the coding stage (I2) indicate a poor final quality index, one feasible action is to increase focus on code inspection of the software modules, allowing fewer defects to escape to the integrated code.

The discrete defect-removal model. Whereas the Rayleigh curve models only the pattern of defect removal, the discrete defect-removal model (DRM) deals with both defect removal and defect injection. The DRM takes a set of error-injection rates by development step and a set of inspection and testing-effectiveness rates as input, then models the defect-removal pattern step-by-step. The defect injection and removal activities at each step (see Reference 10) are described by Figure 4.

The DRM takes a simplified view of Figure 4 and works as shown in Figure 5.

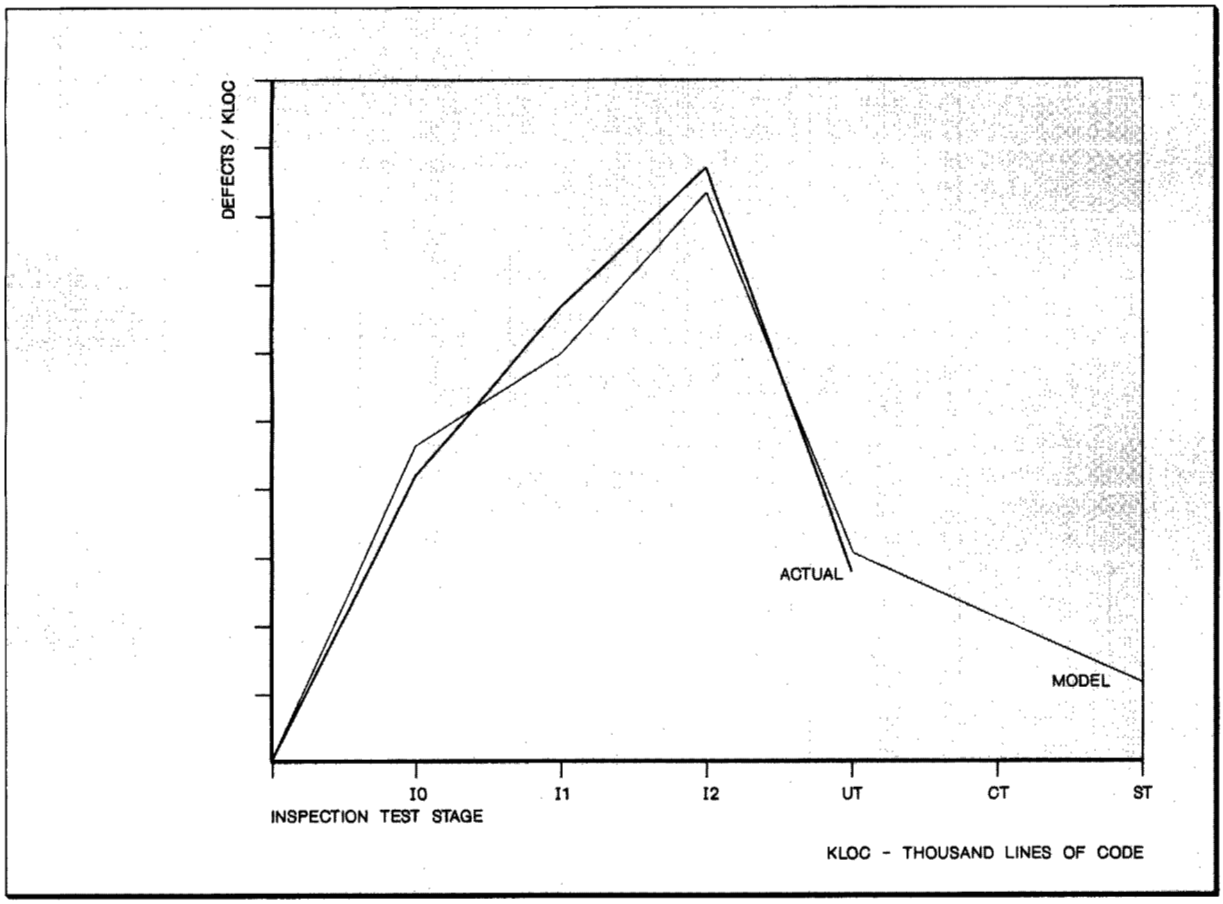
The error-injection rates and the inspection and testing effectiveness are usually based on estimates from the previous release or from historical data. If the total number of defects, or defect rate, of the system are known, the final quality index can be calculated based on the model output at the exit of the last development step. However, unlike the parametric models, the DRM cannot estimate the final quality index. It cannot do so because the total defect rate (or, for that matter, the latent-defect rate when the system is shipped), the very target for estimation, is needed as input to the model. It is a tracking tool instead of a projection tool. The rationale behind this model is that if one can ensure that the defect-removal pattern by step is similar to that for a previous experience, one might reasonably expect to see approximately the same final quality index.

The DRM is a useful management tool which can track current quality status during development

Figure 5 Defect injection and removal step, simplified view

DEFECTS AT THE EXIT OF A DEVELOPMENT STEP	=	DEFECTS ESCAPED FROM PREVIOUS STEP	+	DEFECTS INJECTED IN CURRENT STEP	−	DEFECTS REMOVED IN CURRENT STEP

Figure 6 Discrete defect-removal model



(as shown in Figure 6) and verify the final quality index that is estimated from independent sources.

More importantly, the DRM can be used to evaluate the influence of inspection and testing effectiveness or error-injection rates on the final quality index. If we plan to improve the inspection effectiveness by 15 percent at the low-level design stage, how much can we expect to gain in system quality? If we invest in a defect prevention process and plan to reduce the error-injection rate at the coding stage by 5 percent, how much could

Figure 7 Percent improvement in QI by improvement in inspection effectiveness for an IBM AS/400 product

	INSPECTION TYPE	IMPROVEMENT IN INSPECTION EFFECTIVENESS (PERCENT)			
		2%	5%	10%	15%
HIGH-LEVEL DESIGN	(I0)	0.0%	0.2%	0.4%	1.1%
LOW-LEVEL DESIGN	(I1)	0.2%	1.1%	3.3%	4.3%
CODING	(I2)	1.1%	4.3%	9.8%	14.1%

we gain? Approximate answers to questions like these could be obtained through the DRM. Once the DRM is verified, one can vary the input (for instance, inspection effectiveness of a particular stage) and compute the final quality index again. Figure 7 shows the percent improvement in the final quality index (QI) by improving inspection effectiveness for a software product.

The PTR submodel. For the DRM (as well as for the Rayleigh model), when used for tracking quality status during development, the Rochester Programming Laboratory also uses a submodel to track the Program Trouble Report (PTR) data during the machine testing stages. The PTR is the vehicle for defect reporting and integrating fix when the code is placed under a formal change-control process. Valid PTRs are, therefore, code defects found during machine testing. A PTR submodel is necessary because the period of formal machine testing (part of unit test, component test, and system test) usually spans months, and one must ensure that the chronological pattern of defect removal is also on track. Simply put, the submodel spreads over time the number of defects that are expected to be removed during the machine-testing phases so that more precise tracking is possible. It is a function of three variables:

- Expected overall PTR rate (per thousand lines of code)
- Planned or actual lines of code integrated over time
- PTR-surfacing pattern after code is integrated

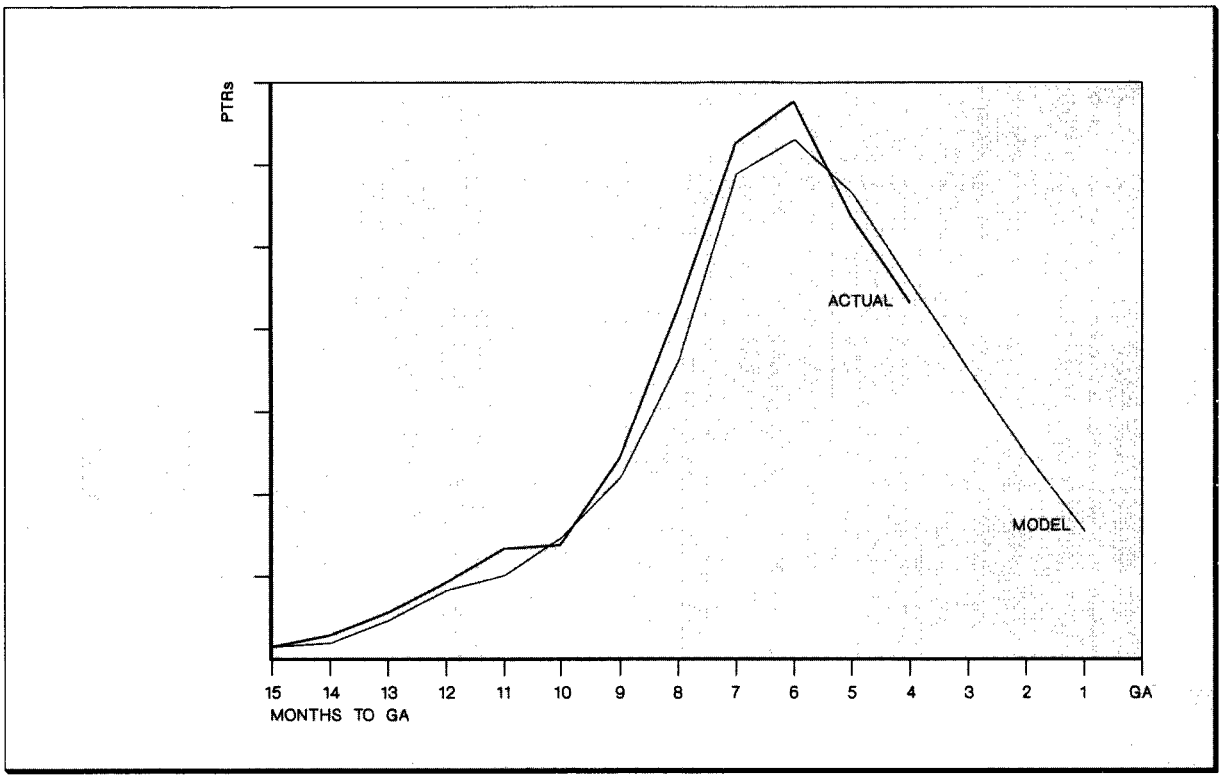
The expected overall PTR rate can be estimated from historical data. Lines-of-code integration over time is usually available in the current implementation plan. The PTR-surfacing pattern after code integration depends on both testing activities and the driver-build schedule. For instance, if a new driver is built every week, the PTR discovery/fix/integration cycle will be faster than that for bi-weekly or monthly drivers. Assuming similar testing efforts, if the driver-build schedule differs from the previous release, adjustment to the previous release pattern is needed. If the current release is the first release, it is more difficult to establish a base pattern. Once a base pattern is established, subsequent refinements are easy. For the IBM AS/400 system, the base pattern was estimated from System/38 data and refined several times during the development process. The current pattern involves a seven-month spread of PTRs after code is integrated.

Figure 8 shows an example of the PTR submodel with actual data. The code integration changes over time during development, so the model is updated periodically. In addition to quality tracking, the model also serves as a powerful quality impact statement for any slip in code integration or testing schedule. Specifically any delay in development and testing will skew the model to the right, and the intersection of the model line and the vertical line of the general-availability (GA) date will become higher.

The PTR arrival/backlog projection model. Near the end of the development cycle, a key question is whether the scheduled code-freeze date can be met without sacrificing quality. Will the PTR arrival and backlog decrease to the predetermined desirable levels by the code-freeze date? The PTR submodel discussed is not sufficiently precise and objective. It is a tracking tool, not a projection tool. On the other hand, the exponential model, or other models based on system-test data, is sufficient, but requires data generated when the system test is well under way. To fill this void, we developed the PTR-arrival/backlog projection model based on the polynomial regression model. Specifically, a few weeks after the start of system test (usually several months before the planned code-freeze date), we modeled the PTR arrival and backlog trend based on polynomial terms of chronological time, time lag variables, cumulative lines-of-code integrated, and other significant dichotomous variables.

This model is different from the exponential model in several aspects. First, the time frame covers all

Figure 8 PTR submodel



machine testing (all PTRs) after the code is integrated (part of unit test, component test, and system test). The exponential model applies only to defect arrivals during system test. Second, the data for this model are PTR arrivals and backlog while the exponential model includes only valid PTRs (defects).

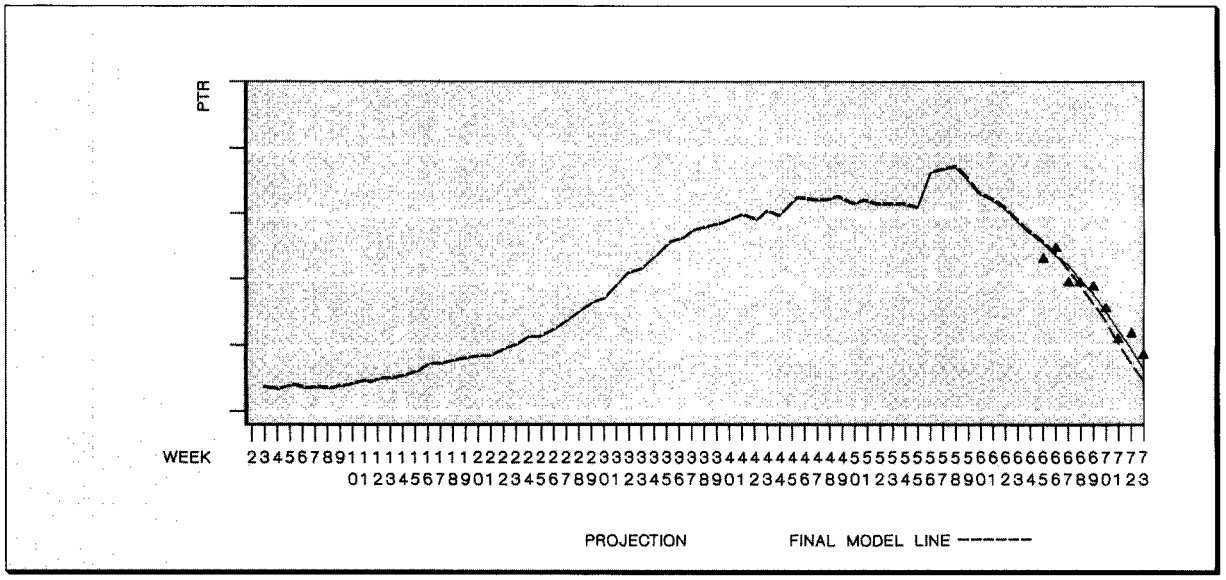
Figure 9 compares the PTR-arrival projection model for AS/400, Release 1, with actual data points for the projection period. The model, with $R^2 = 95.6$ percent, produces a projection that is accurate within a week in terms of when the PTR arrivals would decrease to the predetermined desirable level.

Unlike other models discussed, the PTR arrival/backlog projection model is really a modeling approach, rather than a specific model. Statistical expertise, modeling experience, and a thorough understanding of the data are necessary in order to deal with issues pertaining to model assumptions, variables specification, and final model selection. A desirable outcome often depends on the model's R-square and on the validity of the assumptions. Furthermore, it requires a fairly large number of data points, and the data must pass the last inflection point of the process in order for the projections to be accurate.

Implementation. As mentioned earlier, we implemented the Rayleigh model by SAS programs. GRAFSTAT, an IBM graphic and statistical software system, is also a good tool for models based on such statistical distributions as the Weibull and Rayleigh.

Both the DRM and the PTR submodel are nonparametric and relatively simple: they can be implemented by most programming languages or even by manual calculation. At the Rochester Programming Laboratory, they are implemented in spreadsheet programs. QRISK, a productivity and quality-assessment tool developed by Stott, also implements the DRM (see Reference 11).

Figure 9 PTR-arrival projection model



We implemented the PTR-arrival/backlog projection model in SAS. Since the model is based on the method of least square regression, which is relatively straightforward and robust in terms of computational algorithms, any statistical software can do the job. In constructing the model, however, the assumptions of normality, expected mean error, and homoscedasticity should be examined. The assumption of correct specification should be satisfied if a high R-squared is obtained. In polynomial models, multi-collinearity always exists among the polynomial terms (such as time, t, t^2, and t^3). If multi-collinearity is a concern, centering the values of the variables at their means will yield satisfactory results.

Summary and conclusions

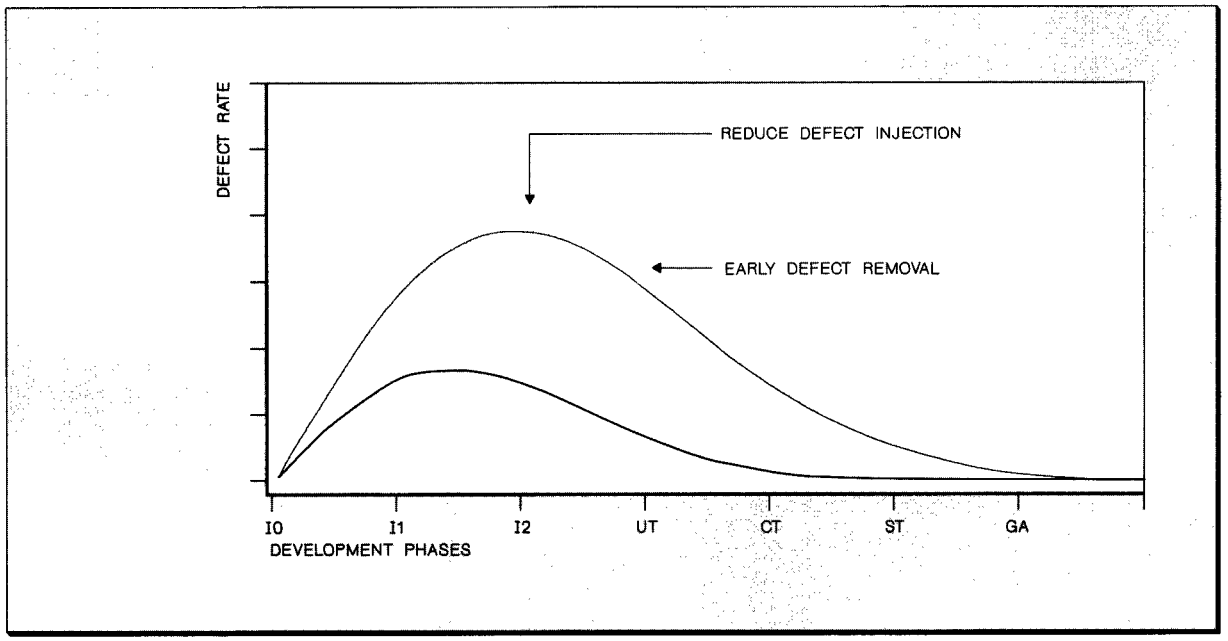
We have briefly summarized the models used at the IBM Rochester Programming Laboratory for managing software development quality and estimating software reliability. The Rayleigh model and the DRM cover the entire development cycle, and the three other models focus on different segments of the continuum of machine testing. All five models have proven useful in the development of the IBM AS/400 software system.

Modeling is a systematic way to describe the complex reality in simpler terms. Because reality changes, the models must be re-examined and updated. We continue to refine our models as soon as new data are available. For instance, the inspection effectiveness of the DRM has been revised, based on the defect data collected during the development of the IBM AS/400. Since the general-availability date of the IBM AS/400 (August 1988), we have been tracking the field defect data and monitoring the predictive validity of the models. Although implementation tools are available, we make sure that every time a model is used, relevant issues are re-examined.

From our experience, the largest obstacle in software modeling was data constraints. For the past several years, we have been making a concerted effort to improve our data collection system. With better data, we anticipate our models will provide better explanations and more accurate projections of the complex reality.

Modeling is closely related to our development quality strategy. Figure 10 shows two key directions in our strategy in relation to the Rayleigh model, which is derived from the AS/400 defect removal data throughout the development process. The first direction is to shift the peak of the curve to the left as much as possible by early defect removal. This can be achieved by process improvement, especially for the design review

Figure 10 Development quality strategy and the defect injection and removal model



and code inspection process. The second direction is to reduce error injection and push the curve downward. This approach is more difficult but also more effective. Actions taken under this approach at the Rochester Programming Laboratory include, among others, specific quality and technical education and laboratory-wide implementation of the defect prevention process. The lower curve in Figure 10 represents our ultimate goal for the defect injection and removal pattern—much lower defect injection and much earlier defect removal.

The role of modeling is crucial to software development quality at the Rochester laboratory. Through modeling, we gain understanding of the mechanics of engineering quality into the development process. In addition to the models discussed in this report, other statistical and quality engineering tools are available. We also take advantage of them whenever the nature of our data permits.

Acknowledgments

I would like to thank Lionel L. Craddock, David J. Lind, and John E. Peterson for their review of and comments on this paper. I am grateful to the anonymous reviewers for their helpful comments.

Cited references

1. A. L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Transactions on Software Engineering* **SE-11**, No. 12, 1411–1423 (1985).
2. P. A. Tobias and D. C. Trindade, *Applied Reliability*, Van Nostrand Reinhold Company, New York (1986).
3. L. H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering* **SE-4**, No. 4, 345–361 (1978).
4. A. L. Goel and L. Okumoto, "A Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability* **R-28**, 206–211 (1979).
5. P. N. Misra, "Software Reliability Analysis," *IBM Systems Journal* **22**, No. 3, 262–270 (1983).
6. M. Ohba, "Software Reliability Analysis Models," *IBM Journal of Research and Development* **28**, No. 4, 428–443 (1984).
7. M. L. Ralston and R. I. Jennrich, "DUD, a Derivative-Free Algorithm for Nonlinear Least Squares," *Technometrics* **20**, No. 1, 7–14 (1978).

8. Personal communications with Michael J. Falcetano and Joseph M. Caruso at IBM Kingston, New York.
9. W. K. Wiener-Ehrlich, J. R. Hamrick, and V. F. Rupolo, "Modeling Software Behavior in Terms of a Formal Life Cycle Curve: Implications for Software Maintenance," *IEEE Transactions on Software Engineering* **SE-10**, No. 4, 376–383 (1984).
10. H. Remus and S. Zilles, "Prediction and Management of Program Quality," *Proceedings of the Fourth International Conference on Software Engineering*, Munich, 341–350 (1979).
11. Personal communications with Dan R. Stott at IBM Kingston, New York.

Stephen H. Kan *IBM Application Business Systems, Highway 52 and NW 37th Street, Rochester, Minnesota 55901.* Dr. Kan is a staff programmer in the Market-Driven Quality department, Application Business Systems Development Laboratory. He holds B.S. degrees in sociology and computer science, M.S. degrees in statistics and sociology, and a Ph.D in demography with majors in projection and estimation, statistics, and survey research. He joined IBM in 1987, and is a Certified Quality Engineer in the American Society for Quality Control. In his current assignment, his focuses are software development quality, AS/400 system software quality plan, and laboratory-wide implementation of the defect prevention process in programming areas.

Reprint Order No. G321-5440.