ELSEVIER

The Journal of Systems and Software 62 (2002) 1–9

The Journal of Systems and Software

www.elsevier.com/locate/jss

The cost of errors in software development: evidence from industry

J. Christopher Westland *

Department of Information and Systems Management, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Received 4 February 2000; received in revised form 13 March 2000; accepted 23 July 2001

Abstract

The search for and correction of errors in software are often time consuming and expensive components of the total cost of software development. The current research investigates to what extent these costs of software error detection and correction contribute to the total cost of software. We initiated the research reported here with the collection of a sample of transactions recording progress on one phase of development of a set of software programs. Each of these projects represented the completion of an identical phase of development (i.e., country localisation) for a different country. This enabled each project to be compared with the other, and provided an unusually high degree of control over the data collection and analysis in real-world empirical study. The research findings relied on programmers' self-assessment of the severity of errors discovered. It found that serious errors have less influence on total cost than errors that were classified as less serious but which occurred with more frequency once these less serious errors are actually resolved and corrected. The research suggests one explanation – that programmers have greater discretion in how and when to resolve these less severe errors. The data supports the hypothesis that errors generate significant software development costs if their resolution requires system redesign. Within the context of the research, it was concluded that uncorrected errors become exponentially more costly with each phase in which they are unresolved, which is consistent with earlier findings in the literature. The research also found that the number of days that a project is open is a log-linear predictor of the number of software errors that will be discovered, implying a bias in error discovery over time. This implies that testing results need to be interpreted in light of the length of testing, and that in practice, tests should take place both before and after systems release.

Keywords: Software errors and reliability; Economics of information technology

1. Software life cycle costs

This research develops a model for costs generated in software development, focusing on costs associated with software error correction and detection. It draws on empirical data extracted for one phase of the software development life cycle by a vendor of packaged microcomputer software.

In the past, empirical investigations of software costs have been rare, most likely due to the difficulty in obtaining sufficient data for analysis. Financial accounting systems do not require separate account information on software error costs and, as noted by Software Productivity Research executive Capers Jones, fewer than 10% of companies who collect software development metrics include defect statistics and even fewer record process errors (Inwood, 1994). Where we do possess knowledge about error cost behaviour, the evidence is spotty and seldom sufficient to build complete models. Inwood (1993) suggested that the norm for software-defect removal is about 75% of the errors that appear in the first year after release. Bloor (1993) found that the number of software errors was most strongly affected by programming quality, software testing and choice of 'safe' programming tools and languages. Boehm (1981, p. 40) (citing prior studies in Boehm, 1980, 1973; Myers, 1976, 1978) suggested that the cost of correcting an error increased exponentially for each phase of the systems development life cycle the correction was delayed. Royce (1993) found that error control costs were positively correlated with the number of 'late-breaking, unforeseen external events'. Violino (1998) found that many IS managers use quality performance measures

* Tel.: +852-2358-7643; fax: +852-2243-0712.
E-mail address: westland@ust.hk (J.C. Westland).
URL: http://www.ismt.ust.hk.

0164-1212/01/$ - see front matter
PII: S0164-1212(01)00130-3

beyond error counts which may depend on the complexity, scale, and importance of the IT project in question. Moser and Nierstrasz (1996) developed software metrics using a statistical correlation between the size of a software project and the amount of effort typically required to realise it. Benyahia (1996) provided a review of the principal costs and productivity estimation method in management information systems, and introduced two new models, testing their sensitivity and reliability. Klepper and Bock (1995) examined third generation languages. Jones (1993) found that faulty scheduling, documentation, schedule pressure, poor training, and 'creeping' user requirements were correlated with higher error rates. Ferneley (2000) investigated the impact of complexity on maintenance costs, and DeLucia et al. (1998) developed software metric tools to evaluate reengineering costs.

The current research is organised into three sections. Section 2 discusses the data collection and control for bias for the empirical study. Section 3 presents a series of pair-wise specification searches that describe the influence of particular cost drivers on costs. Section 4 synthesizes a multiparameter cost model. Section 5 discusses the implications for the software industry, and suggests avenues for extending this research.

2. Data collection

Data was collected from a packaged software vendor with global operations and a number of competitive products. The vendor developed, researched, specified, and tested each software release at corporate headquarters before allowing global release. Each completed, tested, and stable product was then released in the vendors home market simultaneously with its being passed on for localisation (i.e., translation to the local language, culture and practice) to 31 strategically important global markets. The vendor allowed the author to collect data from corporate accounting records and databases on a condition of anonymity. This overcomes perhaps the most significant problem facing prior studies on software error cost models – the commercial sensitivity of the data.

Any analysis of corporate records (such as the current study) tends to suffer from the weakness of other non-experimental sciences – the unavailability of controlled, replicable experiments. The restriction of data collection to activities involved in localisation of an already completed product, as well as the rigorous and formal testing procedure adopted by the vendor, limit potential weaknesses and confounding effects from this non-experimental data. Thus, localisation of software provides a controlled environment for the specification of software cost models. A full release version of the software already exists, and this provides a common 'baseline' for all of the individual localisation projects. This significantly constrains the sources of errors, and assures that projects are comparable. Errors specific to the localisation process arise when converting from the English language software into a particular language and cultural environment. Problems may arise from idiosyncrasies of documentation and presentation; from differing traditions for performing similar tasks required by the software; and from different character sets (e.g., alphabets of English versus characters of Chinese versus bi-directional script of Arabic).

The vendor observed that costs of translation to the local language contributed less than 10% of total cost incurred in the localisation phase of development; the vendor's major 'localisation' expenditure was directed toward error search, detection and correction. Therefore, the incurrence, detection, and correction of errors were the focus of the analysis and model building in this research.

The vendor conducted 31 separate localisation projects; these provide packages for 31 strategic markets. These projects were prioritised into three groups (which are called 'tiers' following the vendor's convention throughout the remainder of the paper).

Local language versions of the software are sequentially released in the three different 'tiers'. Tier 1 versions of the software are released first. They localise versions of the software into the languages of the vendor's four largest markets. Localisation of 'tier 2' begins only after the release of the four 'tier 1' language versions of the software; 'tier 2' covers the next 15 markets. Localisation of 'tier 3' starts towards the end of the 'tier 2' localisation tasks and covers the vendor's remaining markets. Often in these markets, it is economically feasible to localise only portions of the package – e.g., documentation, help screens and menus, but not error messages. A significant amount of learning takes place in the localisation process for 'tier 1'. The faults and idiosyncrasies learned during 'tier 1' assist the localisation of the next two tiers. Typically, there is a delay of one month between the start date of programming on tier 1 and tier 2; and another month delay between tier 2 and tier 3.

The data set contained 21,017 records of error and accounting information obtained from corporate databases and four years of paper reports. These datasets were merged and summarised into 31 observations covering two versions of a single package that were localised for 31 markets. There were 3851 detailed records of errors for the localisation process, each reflecting the contents of error logs prepared by the programmers when they detected and corrected an error. These were restructured into counts or summary statistics in each observation. The vendor observed that the volume of errors detected depends on (1) how many errors exist in total, (2) how hard each is to find, and (3) how intensely,

systematically and intelligently the testers look for errors. The vendor felt that there was considerable variance in each of these three factors across localisation projects especially across tiers.

The firm uses a 'production line' method for finding errors – i.e., a preset number and type of tests (treatments) are made to attempt to produce errors (effects). This approach is systematic, but can be very labour intensive due to the exponential growth in degrees of freedom of testing as complexity of the product grows. For example, if the software needs to be tested on several hundred parameters, for several hundred effects, the size of the test deck can be in the millions. The system for collecting errors is depicted in Fig. 1.

3. Error cost

The vendor's internal studies concluded that the largest single contributor to localisation cost was the detection and correction of errors. Consequently, this research first investigated evidence on error costs.

Prior literature has suggested a growth model for software errors and reliability. Evolutionary growth models assume that the number of errors discovered depends on time and that the underlying error distribution is exponential. The most important of these was proposed by Jelinski and Moranda (1972) and was refined and promoted in Musa and Okumoto (1988), Musa and Iannino (1991) and Musa (1996).

Analysis of the software error dataset collected for this research involved: (1) confirmatory testing to determine whether the evolutionary growth model as presented in Musa and Okumoto (1988), Musa and Iannino (1991) and Musa (1996) accurately represents the occurrence of errors experienced by the vendor; and (2) a specification search to find the error cost model with the greatest explanatory power.

Confirmatory analysis therefore tests the hypothesis H_0 that error count is exponential over time.

H_0: Error count is exponentially distributed in time with c.d.f.:

$$\int_0^T \beta e^{-\beta f(t)}\,dt = -e^{-\beta f(t)} f'(t)\big|_0^T.$$

H_A: Error rate is distributed according to some other continuous, monotonic increasing c.d.f.

To test H_0 the observed error count was re-expressed via a Box–Cox (1964) transformation for various exponents. The Box–Cox transformation re-expresses the error count as powers or a logarithm via the formula $(\text{Errors}^{\lambda} - 1)/\lambda$, where λ smoothly adjusts the shape of the transformation from reciprocal powers for $\lambda < 0$ to the logarithm at $\lambda = 0$ to powers for $\lambda > 0$. Implicit in the assumption of exponential distribution in prior research is the assumption that measurements perhaps have not been taken on a linear scale (not unusual when human judgment is involved). It is useful to search to see if the exponential distribution provides the best fit for the observed data. The Box–Cox transformation flexibly incorporates the so-called 'ladder of powers' into its parameter lambda. Thus if $\lambda = 2$ the function is x^2, if $\lambda = 1$ the function is x and so forth. Table 1 summarises the findings. The transformation at $\lambda = 0$ has the largest F-statistic and R^2. This corresponds to an exponential distribution of error count over time if $f(t)$ is nearly linear, since $\log_e(-e^{-\beta f(t)} f'(t)\big|_0^T) \cong \text{cons.} \cdot t$. This provides strong confirmatory evidence for the adequacy of evolutionary models in describing error occurrence in software.

The tests showed that H_0 is strongly supported, with the following 'goodness-of-fit' measures – an F-statistic of 24.0, R-squared of 55.8; and adjusted R-squared of 53.5. The F-statistic measures the difference between the true model and one where all of the parameters are zero, whereas R-squared is a general measure of the variance in the data explained by the model (it is a percentage between zero and 100); adjusted R-squared adjusts for the size of the dataset. Table 1 shows that an inverse square root function gives slightly better fit. The curve peaks at a value of λ around 0.4, but this particular

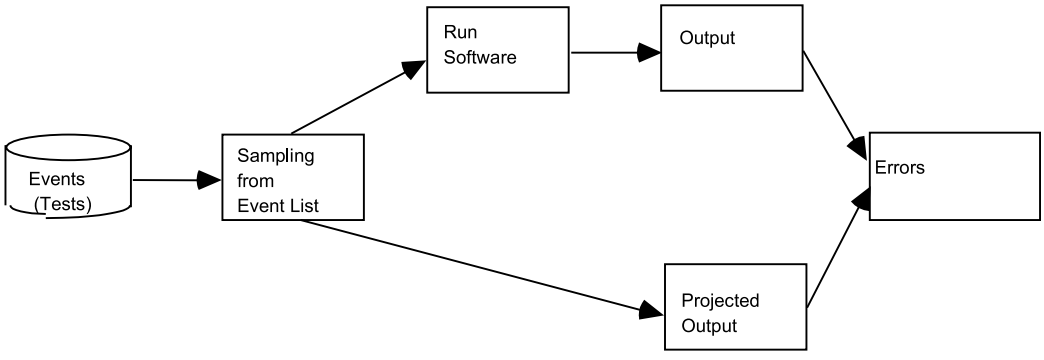
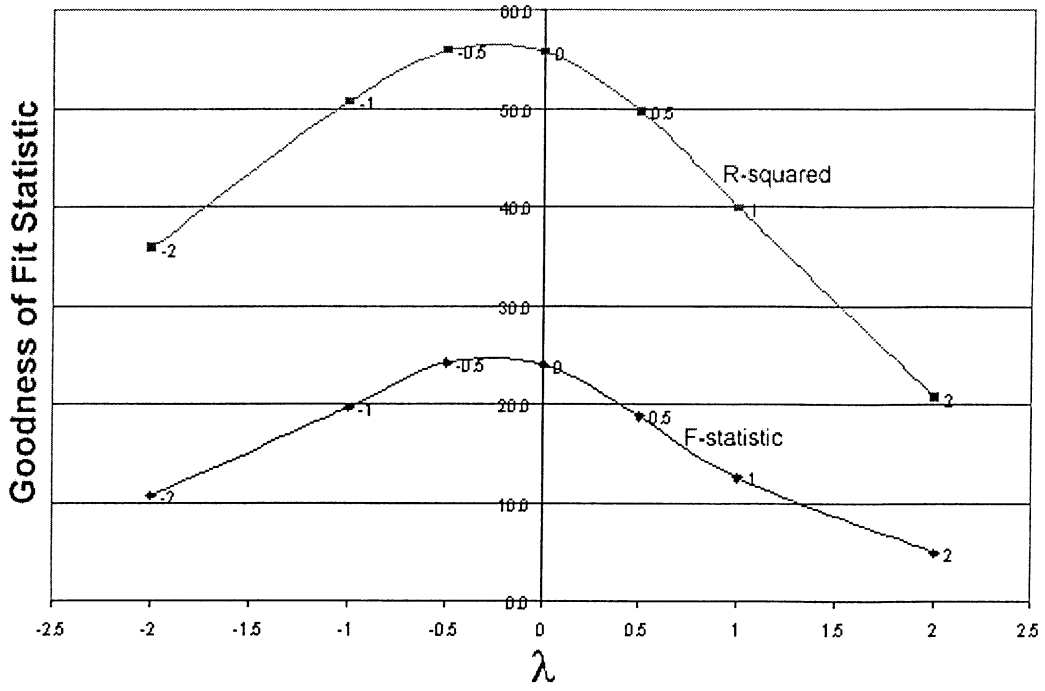


Fig. 1. Process of error testing, discovery and acquisition.

Table 1
Search for f with best linear fit for f (Errors) = Constant + β time



λ	F	R^2	R^2 Adj.	Constant			Time (Project length)		
				Estimate	t-Value	p-Value	Estimate	t-Value	p-Value
2.0 square	5.0	20.7	16.5	1477.16	0.4	0.7289	60.2298000	2.23	0.0383
1.0 raw data	12.6	39.9	36.8	37.65	1.5	0.1451	0.5674300	3.55	0.0021
0.5 square root	18.8	49.7	47.1	10.07	4.5	0.0002	0.0625067	4.34	0.0004
0.0 $\log_e$	24.0	55.8	53.5	3.59	15.2	0.0001	0.0074378	4.90	0.0001
−0.5 – inverse of square root	24.2	56.0	53.7	1.65	58.1	0.0001	0.0008965	4.91	0.0001
−1.0 – inverse	19.7	50.8	48.3	0.97	220.0	0.0001	0.0001264	4.43	0.0003
−2.0 – inverse of square	10.7	35.9	32.6	0.50	3470.0	0.0001	0.0000030	3.26	0.0041

value does not have a clear intuitive significance. It is more meaningful to choose the inverse square root transform $\lambda = 0.5$, with a nearly identical behaviour, and suggest that this has the most explanatory power. The difference between $\lambda = 0.4$ and $\lambda = 0.5$ is well within one standard deviation, and therefore $\lambda = 0.5$ provides a plausible characterisation of error count over time.

4. Specification searches

This part of the research reports the findings from a general multiple variable analysis of software costs and cost drivers for development tasks during the 'localisation' phase. Because part of the data used is accounting cost data, there is the possibility of results being confounded by double-entry bookkeeping. Double-entry bookkeeping, by definition, records information about the same transactions in multiple places around the corporate accounts simultaneously. This can weaken our conclusions due to the multicollinearity of sample data. Multicollinearity is typically a matter of degree, and reflects the weakness of the data for answering the researcher's questions.

To assure that the data obtained was adequate to answer the research questions it was tested for multicollinearity. Principal components analysis was performed for the attributes used in the research to see whether our results were subject to significant multicollinearity. Principal components analysis indicated that the model parameterisation used to answer research questions in the confirmatory and exploratory analysis was appropriate and that significant multicollinearity was *not* present.

4.1. Regression of error occurrence and software cost

When the vendor's test program identifies an error, it is prioritised from 1 to 3:

1. a *serious error* (in the judgment of the programmer) is called *priority 1* and must be resolved;
2. a *moderately severe error* is a *priority 2* error, and may or may not be corrected by release time;
3. a *minor error* which is typically cosmetic, such as spelling, is called *priority 3*.

When priority 3 errors are removed, the regression results improve significantly, suggesting that there may be a good deal of arbitrariness in recording priority 3 errors. These three priority levels are completely independent of the 'tier' designation, which designates the timing of market release. *Tier* is included in the regression to capture differences in the sequence of localizing for different markets.

Dependent variable: Total delivered cost of localised software

	Coefficient	Standard error	*t*-Ratio	*p*-Value
Constant	104 143	24 934	4.18	0.0011
Tier	−23533.7	8955	−2.63	0.0209
Priority 1 errors	−614.322	826.6	−0.743	0.4706
Priority 2 errors	599.089	295.5	2.03	0.0636
$R^2 = 49.0\%$	$F = 4.16$	31 cases		

The regression shows that the count of serious errors is less important in determining cost than the number of less serious errors. This likely reflects greater discretion available in how and when to resolve the less severe errors. Priority 1 errors are likely to need immediate attention, and may be thought of as a fixed cost associated with staffing quality and complexity of localisation.

4.2. Regression of error resolution and software cost

Software errors are costly because the firm uses up scarce resources (mainly programmer labour) in resolving the error. The vendor assumed that there were four possible resolutions to a reported error:

1. The software design is modified to resolve the error.
2. The error has occurred elsewhere, and is resolved in conjunction with resolving the error elsewhere. Resolution of duplicate errors tended to occur within clusters of languages within a tier, especially in the second or third tier where errors may have already been resolved in the first tier, which is localised before other tiers.
3. A decision is made not to resolve the error, e.g., because it is cosmetic, rare, or inconspicuous.
4. The resolution of the error is postponed, typically to allow its resolution in conjunction with resolving the error elsewhere.

The following regression results were obtained.

Dependent variable: Total delivered cost of localised software

	Coefficient	Standard error	*t*-Ratio	*p*-Value
Constant	112 469	27 824	4.04	0.0019
Tier	−24768.6	10 050	−2.46	0.0314
Redesigned	1151.11	815.3	1.41	0.1856
Duplicated	−217.748	2497	−0.087	0.9321
Won't resolve	−170.497	2221	−0.077	0.9402
Postponed	1064.09	1830	0.582	0.5726
$R^2 = 44.8\%$	$F = 1.78$	31 cases		

The *p*-values are unconvincing for the *duplication, won't resolve* and *postponed* actions. This would be expected, since there would be little marginal increase in cost for the *duplicated* and *won't resolve* actions. If the *postponed* action ultimately creates another *duplicate*, then the impact at the margin is also likely to be slight. These results were robust to the selective exclusion of one or more of the variables. Most of the cost arises from errors that require some system redesign.

4.3. Regression of life cycle phase of an error and software cost

Boehm (1981, p. 40) argued that the cost of correcting an error increased exponentially for each phase of the systems development life cycle that resolution is delayed. We can consider the localisation process to have two distinct phases: (1) the creation of the *base software* in the vendor's home market, and (2) *localisation* to a particular global market. Development of the *base software* is assumed to follow the traditional phases of the systems development life cycle, and since it is complete prior to localisation, it is considered as a *single* phase for categorizing the source of errors – errors are either presumed to exist already in the delivered base software or to occur during the localisation phase. The regression results are as follows:

Dependent variable: Total delivered cost of localised software

	Coefficient	Standard error	*t*-Ratio	*p*-Value
Constant	125 079	36 010	3.47	0.0041
Tier	−24290.3	10 748	−2.26	0.0416
Base phase error count	461.928	413.7	1.12	0.2844
Localisation phase error count	−52.5658	198.6	−0.265	0.7954
$R^2 = 33.5\%$	$F = 2.18$	31 cases		

The regression seems to confirm that errors from the earlier phase are much more influential on cost than those for the later phase. The *p*-values for the localisation phase are not very convincing, and the coefficient for the localisation phase is negative implying that errors help reduce costs (which is unlikely). This most likely reflects the overwhelming influence of errors occurring early in the process.

Boehm's assertion that the cost of an error that is unresolved increases exponentially with each additional phase it goes through without resolution can be restated as

$$\text{Software cost} = \beta_0 + \beta_1 \,\text{Tier} + \beta_2 \, e^{\text{Base Software Errors}} + \beta_3 \,\text{Localisation Errors}.$$

The equation characterises the exponential influence of the unresolved, English language *base software phase* errors. A regression gives the following:

Dependent variable: Total delivered cost of localised software

	Coefficient	Standard error	*t*-Ratio	*p*-Value
Constant	134 590	3.026e4	4.45	0.0007
Tier	−22282.3	8560	−2.26	0.0416
Base phase error count	1.04318e−29	3.865e−30	2.7	0.0182
Localisation phase error count	−90.5385	152.6	−0.593	0.5631
$R^2 = 53.3\%$	$F = 4.94$	31 cases		

In the vendors localisation process, the code for the English language *base software* is completed, tested and frozen at the corporate headquarters, and then is sent to specific localisation programming groups to be converted to specific language markets. The base software essentially provides a very detailed requirement and design specification, since all of the functions are detailed in that software. Localisation represents the subsequent development phase – the implementation for a specific market.

This regression strongly supports Boehm's contention that uncorrected errors in the 'base software' become exponentially more costly to correct when they are left until the 'localisation phase'. Anecdotal evidence obtained during this study from programmers suggests that much of the additional expense is accounted for by the search for an error that was not created in the localisation phase. The regression coefficient on the 'base phase error count' is very small because the counts are generally integers between 1 and 100 and thus the exponents become very large. Box–Cox power transformations of base software phase error count across a range of $\lambda \in [-2, +2]$ were investigated, but provided a poor fit in comparison to the exponential transformation.

4.4. Investigation of potential error reporting biases and their impact on software cost

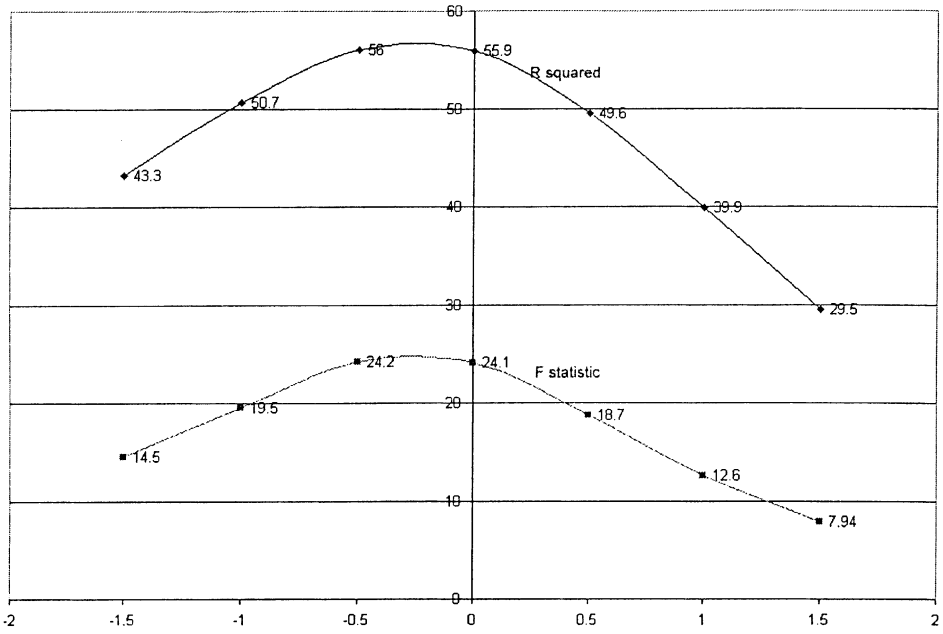
Up to this point, it has been assumed that the number of errors reported is solely a function of technical parameters such as inherent complexity, quality of programming and so forth. However, *reported* errors are also a function of the intensity of search, and the impact of more or less thorough searches on each project need to be taken into account. Localisation of software is performed as well-defined tasks, within a tight schedule, on a base of code that is already in release. The search for errors can begin as soon as the localisation programmers receive the base software. The vendor sets deadlines for compilation, test, and cloture on all programming modifications for a 'build' – after that deadline, the project is designated 'closed' and all subsequent work was to the next 'build' of the software. Because of this, the length of time that a project is open was considered a good measure of the length of time spent searching and the thoroughness of search. Here are the results for a regression of error count on length of time that a project is open.

Dependent variable: Total number of errors discovered in the localised software

	Coefficient	Standard error	*t*-Ratio	*p*-Value
Constant	38.6522	24.78	1.56	0.1353
Days open	0.567430	0.1596	3.55	0.0021
$R^2 = 39.9\%$	$F = 12.6$	22 cases		

The regression indicates that the number of days that a project is open is a good predictor of the number of software errors, which will be discovered. The search for

Table 2
Errors found versus transformed time spent on search



λ	R^2	F-statistic	Constant			Days project was open		
			Coefficient	t-Value	p-Value	Coefficient	t-Value	p-Value
1.5	29.5	7.94	194.966	0.624	0.5398	5.66895	2.82	0.011
1	39.9	12.6	37.6522	1.52	0.1451	0.56743	3.55	0.0021
0.5	49.6	18.7	10.355	4.38	0.0003	0.065741	4.32	0.0004
0	55.9	24.1	3.51474	15.7	Nil	0.007085	4.91	Nil
−0.5	56	24.2	1.64624	58.1	Nil	0.000897	4.91	Nil
−1	50.7	19.5	0.962307	227	Nil	0.000121	4.42	0.0003
−1.5	43.3	14.5	0.664907	864	Nil	1.89E − 05	3.81	0.0012

errors is likely to be non-linear – obvious errors are caught with little effort and cost; other errors require large expenditures of money, time and effort. A Box–Cox transformation yields the results graphed in Table 2.

Recall that the Box–Cox transformation provides a continuous set of 'power' transformations to the data. The curve peaks at a value of λ around 0.2, but this particular value does not have any clear intuitive significance; $\lambda = 0.2$ corresponds to $\text{data}^{0.2} = \sqrt[5]{\text{data}}$. It is more meaningful to choose the natural logarithm transform $\lambda = 0$, with a nearly identical behaviour, and suggest that this has the most explanatory power. Note that the difference between $\lambda = 0.2$ and $\lambda = 0$ is well within one standard deviation, and thus $\lambda = 0$ provides an acceptable interpretation of the data. This supports an explanation that errors tend to be found rapidly at first, trailing off logarithmically as the search continues. The logarithm shows up in many inherently human processes like programming.

As localisation progresses on a given piece of software, participants will learn more about the idiosyncrasies of the product, and thus should find it less costly to localise versions that are scheduled for later release. As noted earlier, particular markets are classified into one of three tiers, depending on their size and strategic importance to the company. The most important markets (tier 1) are localised first, and the least important (tier 3) are localised last. Tier 1 projects are also those that tend to identify significant errors first, and that would be likely to absorb a significant amount of start-up and learning costs. Typically, tier 2 and tier 3 projects stay open longer, because of scheduled completion uncertainties with projects in prior tiers (which are performed by programmers who will eventually be assigned to later tiers). Tier information has been shown in prior regressions to account for a large amount of the variance in project cost. It can also be shown to be a strong predictor of errors discovered, as shown in Fig. 2.

Note that tier 1 regression line slopes down, while the tier 2 line slopes up slightly faster than the tier 3 line. Obviously different things are happening in tier 1 than in subsequent localisation activities. This suggests that software localisation experiences a steep and rapid

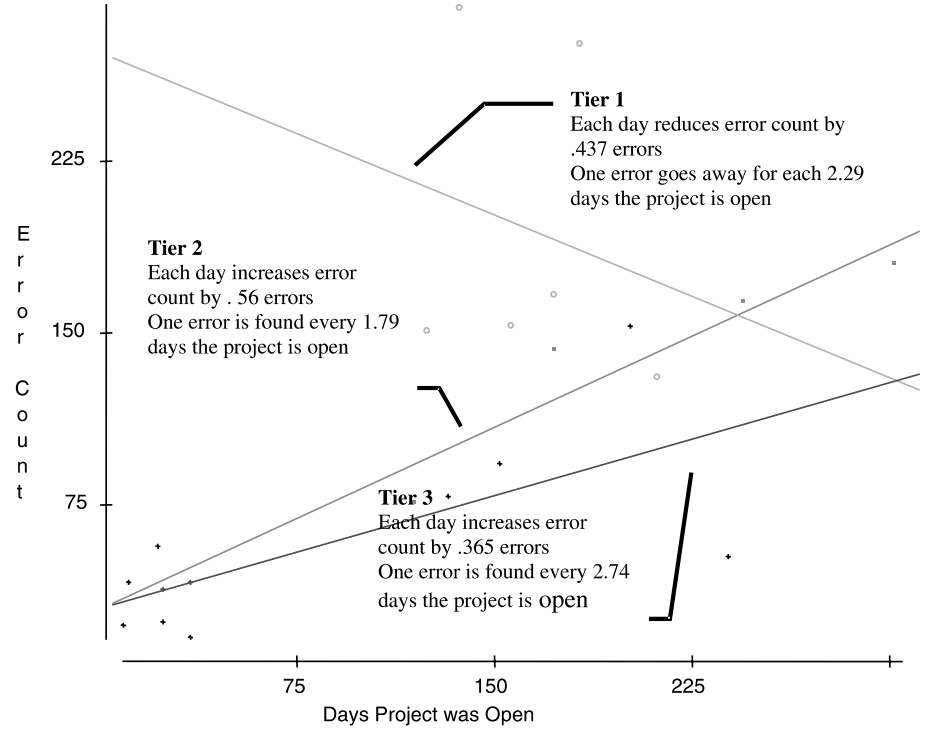


Fig. 2. Effect of tier on error discovery.

learning curve. The longer tier 1 jobs are kept open, the fewer errors are found. Tiers 2 and 3 apparently learn from tier 1's efforts – which is precisely why the vendor chose to split the localisation effort into three tiers.

5. Conclusions and implications for the software industry

The dataset acquired for this research provides strong support for an *evolutionary* model of error occurrence with an exponential distribution of error count overtime. It also provided some additional specifications for enhanced models for software cost estimation. Specifically:

(1) The regression shows that the serious errors have less influence on cost than other errors, assuming that they are ultimately resolved. This likely reflects greater discretion available in how and when to resolve the less severe errors. One explanation is that severe errors (priority 1 errors in the research) are likely to need immediate attention, and may be thought of as a fixed cost associated with staffing quality and complexity of localisation.

(2) Errors only generate significant costs if their resolution requires system redesign. Prior studies have often not considered how errors are resolved only their occurrence. These models are likely to overstate costs, because they fail to discount errors for which no action is taken, and for which there is likely to be no impact on the user after release.

(3) The data support the conclusion that uncorrected errors become exponentially more costly with each phase in which they are unresolved.

(4) The data indicate that the number of days that a project is open is a log-linear predictor of the number of software errors that will be discovered.

(5) The research found that as development progresses on a given piece of software, participants will learn more about the idiosyncrasies of the product, and thus find it less costly to perform tasks scheduled later in the life cycle.

(6) Testing must occur over a substantial portion of the useful life of the system in order to detect a substantial portion of the total errors which will ever occur. Cost effective testing thus should be incorporated into the *maintenance* of the system, with user feedback loops to report on errors when they occur, and to analyse and correct them quickly.

This research has shown that, not only does software error detection and correction contribute a substantial proportion of the total cost of software, but that the management of error detection and correction can be complex. Because human judgments are made throughout the detection and correction process, the cost of the errors may appear either as part of the programming budget, or as increased warranty or user service costs after release. This suggests that the software industry – and increasingly important component of the New Economy – is itself becoming structurally complex. This research has taken an important step

towards understanding the nature of this complexity, in exploring error costs as they contribute to the overall *programming* costs of the software.

Consistent with the *quality assurance* viewpoint adopted in this paper, we could consider there to be fully five classes of metrics to measure the quality of software – *defect*, *technical*, *satisfaction*, *warranty* and *reputation*. Defect measures are not available until the system can be run. Control of defects is the main objective of the testing and debugging phase of software development. Technical measures assess whether the software code is well-structured, whether manuals for hardware and software use are adequate, whether it is complete, correct and up-to-date. Technical measures are available for any system at any time. User satisfaction measures actually describe the value received from using the system. The vendor engages in market studies and beta testing to ascertain user satisfaction. Nevertheless, this data tend to be quite difficult to directly link to development costs, despite the best wishes of the vendor. Warranty costs include technical support and training, and may be one of the most significant costs of software. Warranty costs are influenced by the level of defects, but also by the willingness of users to come forth with complaints, and ability and willingness of the software vendor to accommodate the user. Reputation measures the perceived user satisfaction with the software. Reputation could be significantly different from actual satisfaction for two reasons: (1) because individual users may use only a small fraction of the functions provided in any software package (e.g., consider the fraction of functionality actually used in any word processor); and (2) because marketing and advertising often influences buyer perceptions of software quality more than actual use. This strongly favours a strategy of continuous process improvement.

References

Benyahia, H., 1996. Costs and productivity estimation in computer engineering economics. The Engineering Economist 41 (3), 229–238.

Bloor, R., 1993. Software quality standards: do software quality initiatives such as ISO 9000 reduce the introduction of dangerous programmatic errors? DBMS 6 (8), 10(2).

Boehm, B.W., 1973. Software and its impact: a quantitative assessment. Datamation, 48–59.

Boehm, B.W., 1980. 8th World computer congress developing small-scale application software products: some experimental results. In: Proceedings, IFIP, October, pp. 321–326.

Boehm, B.W., 1981. In: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, p. 1981.

Box, G.E.P., Cox, D.R., 1964. An Analysis of Transformations. Journal of the Royal Statistical Society, Series B 26, 211–243.

Canfora, G., DeLucia, A., Munro, M., 1998. An integrated environment for reuse reengineering C code. The Journal of Systems and Software 42 (2), 153–164.

Ferneley, E., 2000. Coupling and control flow measures in practice. The Journal of Systems and Software 51 (2), 111–118.

Inwood, C., 1993. Formal methods can cut your error rate. Computing Canada 19 (2), 29(1).

Inwood, C., 1994. To err is human, to forgive uncommon in IS. Computing Canada 20 (6), 22.

Jelinski, Z., Moranda, P.B., 1972. Software reliability research. In: Freiberger, W. (Ed.), Statistical Computer Performance Evaluations, Academic Press, New York, NY, pp. 465-484.

Jones, C., 1993. Sick software. Computerworld 27 (50), 115.

Klepper, R., Bock, D., 1995. Third and fourth generation language productivity differences. Communications of the ACM 38 (9), 69–81.

Moser, S., Nierstrasz, O., 1996. The effect of object-oriented frameworks on developer productivity. Computer 29, 45–51.

Musa, J.D., 1996. Software reliability-engineered testing. Computer 29 (11), 61–68.

Musa, J.D., Iannino, A., 1991. Estimating the total number of software failures using an exponential model. SIGSOFT Software Engineering Notes 16 (3), 80.

Musa, J.D., Okumoto, K., 1988. Application of basic and logarithmic Poisson execution time models in software reliability measurement. In: Bittanti, S. (Ed.), Software Reliability Modeling and Identification. Springer, Berlin, p. 1988.

Myers, G.J., 1976. In: Software Reliability. Wiley, New York, p. 1976.

Myers, G.J., 1978. A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM, 760–768.

Royce, W., 1993. Why software costs so much: how to get people and technology to work together. IEEE Software 10 (3), 90(2).

Violino, B., 1998. ROI in the real world. Informationweek (April 27), 60–72.

J. Christopher Westland is Professor and Head of the Department of Information Systems and Management at the Hong Kong University of Science and Technology. He received a BA in mathematics, and an MBA in accounting from Indiana University. He received his Ph.D. in information systems from the University of Michigan. He has professional experience in the US as a certified public accountant and as a consultant in information systems in the US, Europe, Latin America and Asia. He is a US *CPA*, an active member of *Mensa*, and sits on the editorial boards of several of the leading academic journals in information technology and has written two books, *The New Science of Wealth* (forthcoming October 2001, Wiley) and *Global Electronic Commerce* (MIT Press, 2000). He has been a frequent commentator on Hong Kong's technology policy. A full biography and selected papers may be found at http://www.ismt.ust.hk/faculty/westland/.