Pawel JANCZAREK[1], Janusz SOSNOWSKI[1]

Warsaw University of Technology (1)

Monitoring Software Development and Usage

Abstract*. This paper presents the methodology of monitoring software testing and debugging processes during system development and usage. We concentrate on control metrics related to these problems and consider two development models related to practical projects. Basing on the collected data we show the usefulness of the presented approach to control software quality, the effectiveness of development and maintenance processes. We also outline possible improvements in monitoring schemes.*

Streszczenie. *Artykuł przedstawia metodykę monitorowania procesów testowania i korekcji błędów w fazie rozwijania i eksploatacji oprogramowania. Praca koncertuje się na miarach opisujących te procesy w odniesieniu do dwu modeli wytwarzania oprogramowania wykorzystywanych we wdrożonych projektach. Bazując na zebranych danych przedstawiono użyteczność opracowanego podejścia w kontrolowaniu jakości oprogramowania oraz efektywności procesów jego wytwarzania i utrzymania. Wskazano również możliwości poprawienia efektywności procesów monitorowania. (**Monitorowanie rozwijania i eksploatacji oprogramowania**).*

Keywords: monitoring bug reports, software reliability, software metrics, testing.
Słowa kluczowe: monitorowanie raportów o błędach, niezawodność oprogramowania, metryki, testowanie.

doi:10.12915/pe.2014.02.31

Introduction

Software reliability is still a challenging problem for software development firms. There is a very reach literature devoted to various aspects of this problem. It covers specification of development and maintenance processes [1], various metrics and models in software quality engineering [2], sophisticated reliability modelling and improvement techniques [1,3-5], etc. They can be treated as guidelines or even standards for some application domains for developers, testers, etc. Many of these techniques involve monitoring various measures, analysing their impact on quality factors and on development or maintenance effectiveness. In practice, direct usage of the proposed techniques faces some limitations.

In general, we distinguish control and product software metrics. Control metrics are associated with software processes, e.g. number of detected defects, defect repairing times, defect severity, testing times, program changes, updates, etc. Product metrics characterize software complexity (e.g. cyclomatic complexity, lines of code, number of classes and associated methods, length of identifiers, depth of nesting). Product metrics can be useful in performing various predictions, e.g. the required person days needed to develop a system component. Additionally we can introduce operational profile metrics characterizing usage of the software. All these metrics can be used in project management decisions and quality evaluation (identification of bottlenecks in processes, problematic modules, prediction of needed resources to achieve the final goal, etc.).

Unfortunately, practical results of software metrics and their analysis related to real projects are rarely encountered in the literature. Moreover, various companies use different development and testing schemes or policies. Developed projects may have stable or changing specifications and are based on various technologies, e.g. developing own programs or integrating commercial components. All this has a big impact on the interpretation of monitored measures. We faced these problems in several projects.

The goal of this paper is to present our experience in monitoring different classes of projects. It is based on the available data which was collected according to development policies (mostly control metrics). Nevertheless, this allowed us to evaluate the practical significance of this data and derive possible extensions or improvements. In the considered projects we had a wide access to the so called problem reports. Here, we faced development schemes not consistent with classical software reliability modelling assumptions. However, they can be considered as typical for many developers.

The paper outlines the scope and policies of problem reports. This is followed with some statistical and analytical results. Basing on these results we discusses possible enhancements, such as inclusion of event and performance logs. Final remarks are given in the conclusion.

Control metrics in software life cycle

Most software development companies monitor the effectiveness of the involved processes and product quality using commercial or their own tools. Typically, they assure dynamic collection of some selected metrics during program testing, debugging and after the system has gone into use. The collected data is introduced in a more or less structured form of reports generated by people involved in various processes (e.g. testing, defect repairs, users, administrators, managers). Unfortunately, in many cases too much freedom is allowed in this reporting, which results in hidden information or imprecise data accuracy. An important issue is to correlate this data with system releases and versions.

In our considerations we concentrated on testing, debugging and maintenance phases of software life cycle. Our experience relates to two models of software development: A – development phase followed by testing, debugging and then operation combined with maintenance, B – incremental development with interleaved phases and releasing partial solutions (e.g. comprising specific functions) to the users. In both models testing is performed by people not involved in development. In the case of model B we can also distinguish the stabilized phase of operation and maintenance (related to the final release comprising all functionalities). We take also into account a third model C which is related only to operational and maintenance phase of stable (matured) projects.

In general, the documentation of the realized processes within the considered models may differ on the accuracy of reports (time stamp resolution, comprised information, sources of the reports, etc.). These reports can be used to derive appropriate software metrics. In the simplest case we have reports specifying time stamps of detected faults (possibly with the severity level). This is typical to most software reliability growth models described in the literature [3,5-7] and it is consistent with our model A.

In the case of model B problem reports are much more informative and accurate. In particular, they comprise the following data: problem ID, ID of problem provider, system

release and module ID, problem registration time stamp, problem severity, release window (RW), problem detection environment (testing, production phase), problem solution progress, problem closing date, problem closing reason, problem description.

In the analysed systems 5 severity levels have been distinguished (user oriented approach): PS1 - the lowest level related to cosmetic defects (they do not influence system functionality, mostly relate to some inconveniences), PS2 – functional minor problems which can be overcome by the users (involving more laborious activities, e.g. based on other available functionalities), PS3 – relates to some functionality problems which create significant inconvenience for the users, PS4 - relates to significant functionality problems which cannot be handled by the users, PS5 – critical problems related to unavailability of basic and fundamental functions, the consequences are severe, e.g. business losses. Depending upon the goal of analysis we can also introduce other categorization, e.g. recoverable defects, non-recoverable defects with various levels of losses (short term or long term system crashes, data or control losses, financial losses such as incorrect billings, wrong account states).

Reports related to model C can be considered as some subset of those for model B. In particular, they are less accurate and can provide basic information (from the user point of view) on problem appearance and resolution times.

Problem reports can be correlated with test schemes (test cases). They can be attributed to individual testers and characterised by execution reports comprising starting and termination time stamp supplemented with the test result: passed or unsuccessful (optionally specified ID of the registered problem). The derived dynamic metrics can be referred to some static metrics, e.g. planned time schedule of release windows, scheduled test advancement.

The derived metrics can be used to evaluate system reliability, project progress, identification of development bottlenecks or risks, reallocation of resources to achieve successful goal. Interpreting derived metrics it is reasonable to take into account the context of their registration and collection. In particular, we should be conscious of the execution profile (during testing or operational phases), problem registration schemes (accurate individual or periodical summarized reports), the number, engagement and skills of information providers.

In the sequel we give a sample of monitoring results for the 3 models related to software projects handling data bases and outline the appearing problems, in particular inconsistencies with classical analysis models.

Analysing monitoring reports

Analysing test and problem reports of some commercial software projects we checked the possibility of deriving some features characterizing reliability (quality) issues as well as the effectiveness of the related software production and maintenance processes. This analysis is referred to 3 project models (compare the previous section).

In the case of model A the available data was limited to precise dates of detected problems during testing phase and problem severity. So practically it allowed us to derive software reliability growth models (SRGMs). These models evaluate the improvement of program reliability in function of testing results (e.g. time between subsequent faults, number of faults in time intervals). In the literature many SRGM models have been proposed with different assumptions [3-5], quite often they are not consistent with real software development schemes. In particular, they estimate m(t) - expected number of errors detected by time t basing on a history of error detection till some time t_a<t.

Basing on test data of project A (cumulative number of detected errors in subsequent days - fig. 1) we have derived several SRGM models compatible with the test scenario and evaluated the predicted total number of faults in the system in the range [547-582]. For an illustration we present one of these models - S-shape Pham Nordman [3]:

(1) $m(t) = \{\alpha[1-\exp(-bt)](1 - \alpha/b) + \alpha at\}/(1 + \beta\exp(-bt))$

the derived parameters are as follows [8]: a=310.94, b=0.077, α= 0.0068, β= 11.92. Hence, for $t\rightarrow\infty$ we get 550. In fact, after 2 years of using this application total number of registered faults was 576, so the prediction was quite good and confirmed that 113 days of testing was reasonable. It is worth noting that considered S-shape models take into account so called learning phase of testing, which was important in this project (simple models, e.g. Goel Okumoto [8] provided wrong predictions over 800 faults).

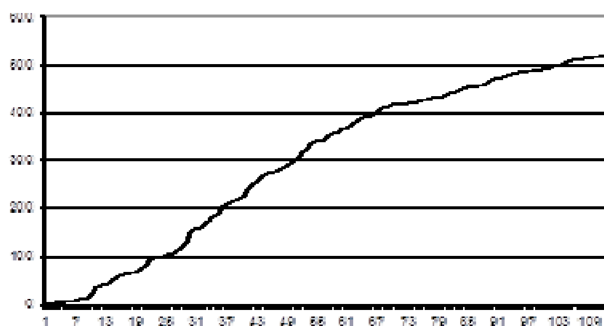


Fig.1. Cumulative number (y axis - [0-600]) of detected errors for 113 days (model A)

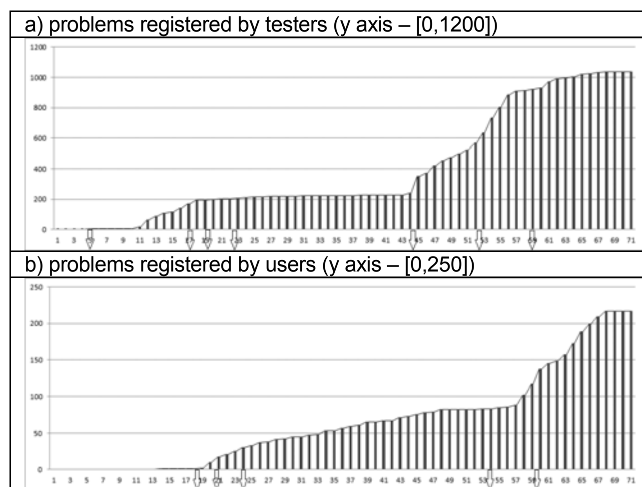


Fig. 2. Cumulative number of registered problems within 71 days (model B)

In general, it is not easy to satisfy assumptions of popular SRGM models (e.g. immediate fault repair, stable testing rate, perfect debugging [9]). Recently, interleaving development with testing and operation phases occurs quite frequently, moreover we have access to more interesting data (e.g. time of resolving problems). Hence, more comprehensive analysis can be performed. This is the case of model B (see the previous section). Here, deriving SRGM models is much more complicated. In fact, we have a common repository of problems detected by testers and users, moreover they may relate to different system revisions. For an illustration in fig. 2 we give plots of the cumulative number of registered problems (within 71 weeks) filtered out for testers and users. A combined summarized plot could be intractable. We can observe shape fluctuations in the plots

related to subsequent releases (marked with arrows), moreover users deal with tested revisions so appropriate time shift of the two plots is visible. Hence, in practice we have derived separate SRGM models for testers, users and releases. They allowed us to evaluate the reliability features.

The relatively reach problem repository allowed us to derive more interesting features characterizing the production processes. In particular, we could derive the number of open problems (needing debugging) in time periods (e.g. weeks). This feature illustrates the load of debuggers, for the considered project we give this in fig. 3. Here, we see some increasing trend at the end, the project manager activated more debuggers to resolve the appearing bottleneck.

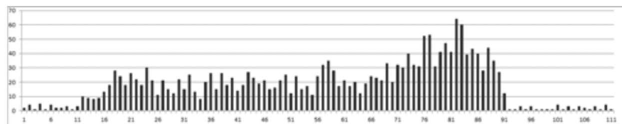


Fig. 3. Open problems (0-70) registered by the users in 111 weeks

Dealing with debugging we are more interested in problem resolution time. In tab. 1 we give some statistics of problem resolution for one of the projects of model B. This statistics relates to lowest (PS1) and highest (PS5) severity problems (reported by testers).

Table 1. Problem resolution time statistics

days	0-10	10-20	20-30	30-50	50-70	70-90	>90
PS1	454	213	165	94	35	19	8
PS5	52	22	13	25	23	5	3

Most problems have been resolved relatively fast (several days). Nevertheless, some needed much more time or were treated as less important. Moreover, some have been handled with long delays by external providers of imported components.

Distribution of problem severity may differ upon the system. This is illustrated for a sample of 11 systems in fig. 4. It is worth mentioning that problems reported by testers usually have higher percentage of high critical problems than those reported by the users (they deal with debugged versions).

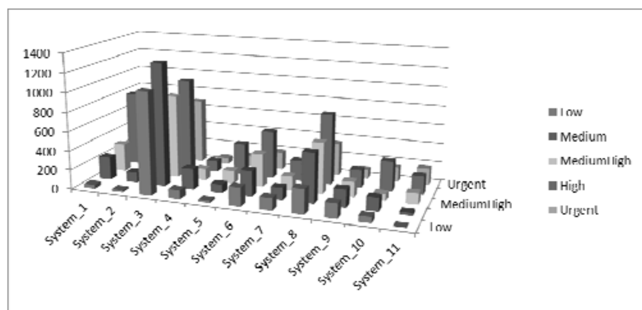


Fig. 4. Fault severity distribution for 11 systems

In practice, we can monitor not only problem reports and their solution progress but also details of test execution. In particular, we can trace the effectiveness of problem detection for various test approaches e.g. module tests, integration tests, acceptance tests, etc. Further, we can trace the contribution of each tester or user (however this should be correlated with their activity). In some projects testing process is carefully planned with predesigned test scenarios comprising explicitly specified test cases. Here, the project manager can check the difference between the scheduled and performed test advance (in per cent). In fig. 5 this is illustrated for one test scenario of a project. There is some difference between the scheduled (upper plot) and performed tests (lower plot). Within the performed tests we can also distinguish unsuccessful tests i.e. such that could not be executed due to some inconstancies of the environment, etc. Moreover, we can display also the number of passed and non-passed (problem detected) tests. Please note flat periods in the plots of the scheduled progress related to weekends.

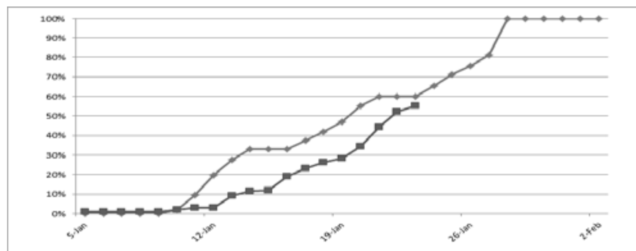
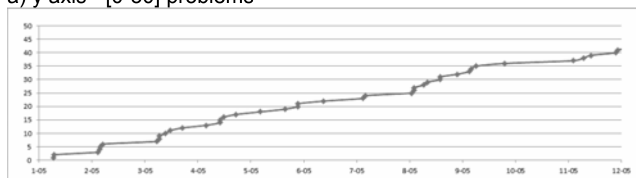


Fig. 5. Monitoring test progress (0-100%) within 5 weeks

Monitoring maintenance phase (model C) is an important issue due to serviceability and warranty problems. Here, we can use similar problem repository, however it is filled by the users. Moreover, the frequency of appearing problems is much lower than during the development phase. We can also derive software reliability growth models assuming that the registered problems are resolved (eliminated) by the software provider. However, the collected data is usually not so abundant as in the previous phases. An important issue is to trace problem resolution time. We illustrate this for some project with data on registered problems and their resolution time (fig. 6). From the user point of view more interesting is the distribution of problem resolution time. The more, that it can be an issue of warranty agreement e.g. specification of maximal waiting time for problem resolution. In practice, we can have projects with various service quality levels. In tab. 2 we give distributions of problem resolution for 3 projects of different service levels (SL1 – SL3: with SL1 the lowest level i.e. the highest service time). Here, we have distinguished 6 ranges of service time and gave the number of attributed problems (NP) related to appropriate ranges.

a) y axis - [0-50] problems



b) y axis - [0-0.3] days

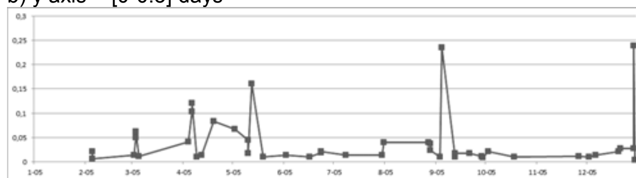


Fig. 6. Maintenance statistics for 12 months: a) cumulative number of registered problems, b) problem resolution time

Having analysed reports of problems during maintenance of a matured product we have noticed relatively low frequency of problems. Moreover, most problems have been resolved by some administration and configuration actions specified by the service desk, so the response time was really short. More severe problems needing code modifications were very rare and primarily resolved by intermediate ways.

Table 2. Distribution of service times (NP – number of problems)

SL1		SL2		SL3	
days	NP	hours	NP	hours	NP
[0-2)	60	[0-2.4)	44	[0-1.2)	35
[2-4)	22	[2.4-5.0)	6	[1.2-2.4)	4
[4-6)	10	[5.0-7.5)	2	[2.4-3.6)	2
[6-8)	3	[7.5-14)	0	[3.6-5)	1
[8-10)	2	[14-16.5)	1	[5- 6)	2
[10-12)	2	[16.5-19)	1	>6.0	0

Enhancing monitoring schemes

The gained experience with several software projects performed according to different development and testing schemes revealed drawbacks of used monitoring policies. They relate to report accuracy, lack of metrics characterizing profiles of testers or users and their engagement, lack of metrics characterizing project complexity starting from the requirements specifications to distribution of code sizes on modules, versions, updates, etc. Moreover, some metrics of debugging complexity (e.g. per cent of modified code) could be added. In the analysis we have to take into account planned revision windows and unplanned (asynchronous) updates. The problem reports should specify clearly transition moments of different phases of problem handling (e.g. diagnosis, resolving).

Analysing the contents of problem descriptions we have found that they are specified in a very free form, mixing Polish and English words, technical jargon, hermetic texts difficult to understand by people not knowing the tested system in details. This makes difficult automatic text mining to classify problem symptoms and solution methods which could be useful in developing service desk repository. Hence, more formalized problem reports are advisable. Quiete often diagnosing (revealing) the source of problem required exchanging emails or telephone calls with the testers and users, even more some screen shots or excerpts from application logs are included. This information usually is skipped in reports, however some metrics of this discussion could be defined and included to assess the complexity of the problem.

Performing testing we usually rely on different test methods, operational profiles and the complexity of test cases. Combining this knowledge with test monitoring may support project manager decisions e.g. having identified a significant delay in the test schedule plan some redistribution of human resources or overtimes can be involved. This has an impact on fault detection intensity. Hence, the derived prediction models should admit appropriate corrections and additional coefficients.

We should also take into account available system logs, e.g. event and performance. They are especially useful during operation and maintenance phases. They provide some information on interaction of the analysed system with the environment, changes in hardware or operating system configurations, upgrades, administrator activities, etc. They should be correlated with classical problem reports. The more that recently complex software and hardware environment may create many problems with the application operation. We have got much experience in this area also. This is reported in [10-12].

In many commercial projects we have some restrictions imposed by the clients on monitoring the installed software due to data sensitivity. Here, some intermediate and synthetic metrics can be used (e.g. instead of the real number of active customers and used resources other relative metrics). Another possibility is value and time scaling of metrics. This has to be agreed with the software provider and owner.

Conclusion

Having analysed problem reports related to developing real software projects we have found that they can provide valuable data to control development, testing and maintenance processes. On the other hand the accuracy and information contents of reports can be improved. Moreover, for better result interpretation it is reasonable to trace various features describing the above mentioned processes (neglected in practice). Another observation is the need of extending monitoring problem reports with system logs. This is especially important during software maintenance phase [13,14]. The presented results were based on real projects and our analysis is ex post. However, the conclusions attracted the related companies to extend and improve monitoring schemes. The considered model with interleaved development and usage phases is typical for some projects (not covered in the literature).

Further research is targeted at correlating monitored features with program requirements, structures, workload profiles, human resources, etc. This can facilitate project management (compare [15]).

REFERENCES

[1] Sommerville I., Software engineering, 9th edition, Pearson, (2011)
[2] Kan S. H., Metrics and models in software quality engineering, Addison Wesly, (2003)
[3] Pham H., Software reliability, Springer, (2000)
[4] Zeng J., Li J., Zeng X., Luo W., A prototype system of software reliability prediction and estimation, *Proc. of 3rd Int. Symposium on Intelligent Information Technology and Security Informatics,* (2010), 558-601
[5] Bluvband Z., Porotsky S., Talmor M., Advanced models for software reliability prediction, *Proc. of IEEE Annual Symp. on Reliability and Maintanability*, (2011)
[6] Chen Y-Ch., Wang X. W., Neural network based approach on reliability prediction of software in maintenance phase, *Proc. of IEEE IEEM Conference*, (2009), 257-261
[7] Huang Ch-Y., Lyu M. R., Estimation and analysis of some, generalized multiple change-point software reliability models, *IEEE Transactions on Reliability*, 60, No. 2, June (2011), 498-511
[8] Sosnowski J., Sabak J. Software reliability analysis in designing data base oriented applications, *Proc. of 27th Euromicro Conf. IEEE Comp. Society*, (2001), 166-173
[9] O. Krini, J. Borcsok, New scientific contributions to the prediction of the reliability of critical systems which base on imperfect debugging, *Proc. of International Symp. on Telecommunications, IEEE*, (2012)
[10] Król M., Sosnowski J., Multidimensional monitoring of computer systems, *Proc. of IEEE Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, (2009), 68-74
[11] Sosnowski J., Kubacki M., Krawczyk H., Monitoring event logs within a cluster system, *Complex Systems and Dependability (ed. W. Zamojski et al.), Advances in Intelligent and Soft Computing*, Springer, 170, (2012), 259-271
[12] Latosiński P., Sosnowski J., Monitoring dependability of a mail server, *Przegląd Elektrotechniczny (Electrical Review), Sigma NOT*, No 10b, (2012), 223-226
[13] Yu Li, Zheng Z, Lan Z., Practical online failure prediction for Blue Gene/P: Period-based vs. Event-driven, *Proc. of the IEEE/IFIP International Conference DSN*, (2011), 259-264
[14] Salfiner F., Lenk M., Malek M., A survey of failure prediction methods, *ACM Computing Surveys*, 42, No. 3, March (2010)
[15] Ferrucci F., et al., Not going to take this anymore: Multi-objective overtime planning for software engineering projects, *Proc. of ICSE Conference*, (2013), 462-471

***Authors**: mgr inż. Paweł Janczarek; prof. dr hab. inż. Janusz Sosnowski, Politechnika Warszawska, Instytut Informatyki, ul. Nowowiejska 15/19, Warszawa 00-665,*
E-mail: Pawel.Janczareki@gmail.com; J.Sosnowski@ii.pw.edu.pl