INFORMATION
SYSTEMS IN
MANAGEMENT

Information Systems in Management (2015) Vol. 4 (3) 171−182

MANAGING COMPLEX SOFTWARE PROJECTS

PAWEŁ JANCZAREK, JANUSZ SOSNOWSKI

Institute of Computer Science, Warsaw University of Technology

In the paper we present our experience with development and maintenance of complex software systems. In particular, we concentrate on monitoring related development, testing and debugging processes. We have analyzed the contents of collected reports (provided by different tools) covering many projects and defined several metrics and statistics helpful in managing complex projects and achieving high quality software. Moreover, we have identified lacking data which could improve these processes.

Keywords: Software Development and Maintenance, Testing, Project Monitoring, Data Analysis

1. Introduction

Contemporary information systems are becoming more and more complex in software. They are characterized by long time development involving many engineers (developers, testers, debuggers, project management) followed by software maintenance. In big projects it is important to manage and monitor all these processes. There are various standards and methodologies providing general rules on how to deal with these processes [11, 12, 17]. In particular, they underline the need of monitoring various aspects characterizing the progress, effectiveness and quality of the involved processes. For this purpose various commercial or open source tools have been developed to collect data on the project progress, they are useful in project management decisions. There are many publications devoted to specific problems. Most of them deal with the flow of development processes at some ab-

stract level, e.g. workload within different development phases [13], prediction of reliability or other features (based on derived analytical models) [3, 5, 14, 18].

Having gained some experience with commercial big projects we present the capabilities and limitations of monitoring development and maintenance processes in relevance to typical data repositories (test progress, problem handling reports) created and managed in such projects. We give an outline of related problems considered in the literature and confront them with our experience. In general, we have observed abundant information in the created repositories which results from more complex models of development and debugging as well as from the capabilities of the used supporting tools (e.g. Redmine, TestLink, Mantis Bug Tracker, TRAC). We concentrate on monitoring the progress of testing and handling detected problems during system development and maintenance (exploitation). Having analyzed various collected data from real projects targeted at services in telecommunication domain (class of CRM systems) we propose various analyses schemes which are helpful in project management. They provide more accurate evaluation of problem handling processes, resulting in quality improvement of software projects.

Section 2 systemizes problems of project monitoring. Section 3 presents the main features of available tools and related data repositories. The range and usefulness of data analysis are illustrated in section 4. Final conclusions are briefly summarized in section 5.

2. Project monitoring problems

Software project development involves product specification, design, implementation and testing. The resulting software product is thereafter delivered to the end users or the market and then maintained. In complex projects all the related activities are time consuming, distributed among many actors and they have a big impact on product quality and its cost. Hence, an important issue is to assess these processes in order to identify bottlenecks (inefficiency) of the processes and introduce necessary improvements or corrections.

The assessment process needs collecting appropriate characteristic data on performed activities and their effectiveness. In the literature there are many studies related to these problems, however most of them rely on repositories storing failure detection times [14-16, 19]. These data are useful in so called software reliability growth modelling (SRGM). SRGM models are derived using recorded error detection times and they provide the capability of assessing test effectiveness and product reliability, e.g. the needed test time to achieve the specified reliability level, the number of the remaining errors [14, 18]. Some enhancements can be included to distinguish several failure severity levels, non-perfect corrections [10], testing effort changes [4], etc. ([15]). This is product oriented analysis. Another approach is targeted at assessment of development and maintenance processes. Here, we can

trace efficiency and work impact in different processing phases (e.g. designing, testing, problem analysis, correcting, retesting, deployment). To characterize these processes various indicators or measures and associated interpretation can be proposed in relevance to different abstraction levels, e.g. completion time of tasks, failure detection and correction rate, failure handling progress within different stages, testers effectiveness, users activities, upgrade changes, etc. The introduced measures can be categorized, e.g. describing customer satisfaction, development and maintenance costs, usage of human and technical resources, correlation with assumed time schedules.

Some interesting papers based on case studies deal with cumulative flow diagrams (CFDs [13]). They allow to detect partially done work, bottlenecks, discontinuities in workflow, excessive hand-overs, waiting and service (problem handling) times, etc. The y-axis of the CFD shows the cumulative number of problems (tasks, requests) in relevance to time (x-axis). CFDs comprise several increasing line plots each corresponding to the appropriate phase. The top-line relates to the total number of inflow problems, the second line from the top presents the handover from the first phase to the second one, etc. For specified time moment x the distance between the top line and the succeeding one presents the work in progress (expressed in the number of problems) in phase 1, etc. CFDs illustrate flow continuity and throughput. Line flatness or low upslope as compared with an upper line relates to continuity problems, e.g. longer inactivity periods which may later result in some overload (typical for integration testing phase). The throughput problem arises while the handover of phase i is higher than in the phase $i+1$ (bottleneck situation of more problems flowing in than out).

CFDs are useful in the case of a small number of phases (problem handling states). They are not satisfactory in the case of more states, moreover they do not show loopbacks, which we have identified as an non negligible effect in real projects. Hence, we have developed more sophisticated graph model PHG (problem handling graph) described in [7]. The nodes correspond to problem handling phases (states) and edges describe transitions. This graph is correlated with a data base describing characteristics of problems and handling times for each problem.

To derive product or process oriented metrics we have to collect appropriate data during development and maintenance. This is the basis for statistical process control (SPC) which is helpful to optimize processes and assure high product quality [5, 13, 17]. Many software companies improve their development processes according to CMM, CMMI or other concepts [11, 12], however SPC approach is rarely applied. Quite often software development companies use various tools to create data repositories relevant to these processes. In practice, they comprise a lot of data which is neglected or not analyzed in a systematic and formalized way. On the other hand some important data is not collected. In the literature the information contents was neglected and most papers used only some selected data.

We have analyzed various repositories related to real long time projects and tried to identify interesting data for management authorities. In the sequel we outline the capabilities of collecting such data using some popular tools. As compared with other approaches discussed in the literature we deal with abundance of data in problem report repositories and perform fine-grained analysis.

3. Tools for collecting data

There are many tools (commercial and free) supporting software project management. They are used in reputable or big companies and less popular in small software companies. These tools are targeted at different aspects, e.g. monitoring project progress in relevance to release deadlines, monitoring testing or problem handling activities. Usually, they provide a lot of data stored in some form of repositories, handle multiple projects simultaneously and provide rich GUI interfaces accessible via web browser. The structure and contents of these repositories can be configured. In this section we give an outline of possibilities of some popular tools which are used by many software development companies.

TestLink is a tool (http://www.testlink.org) designed for test case management. It provides a centralized repository for managing requirements and tests for project/system. It supports all testing stages (requirements specification, test planning, preparation of test scenarios, test cases). It assures flexible management of user roles and provides reporting on test execution (including visualization of reports, related metrics, statistics, graphs generation), which is useful to monitor progress of test cases or scenarios. It can trace the implementation of test cases for many environments (e.g. testing, development, production). It can cooperate (exchange information) with error/problem management tools (e.g. Bugzilla, JIRA, Mantis BT).

TestLink repository stores data on tester activities and roles, test cases and scenarios, test plans and assignment of test cases (time scheduling), test results, etc. The whole repository comprises about 60 tables, which cover all these data. This repository can be adapted to the requirements of the project - for example specific attributes.

The main entities used by TestLink are: test case, test suit, test plan, test project and tester (user). Test case describes a testing task using steps (actions), and expected results. Test cases are the building blocks of TestLink. Test suite groups test cases into units. It arranges test specifications into logical parts. Test plan is created for test cases and specifies their execution time schedule. Each test plan can include releases (builds), milestones, user assignments and finally test results. Test project consists of test specification with test cases, requirements and keywords. Test project is a persistent object through lifetime of the project in TestLink.

Each TestLink user has assigned roles, which grant available TestLink features to this user. Tester and quality assurance (QA) leader, can create test cases, run those test cases, save results or adjust them (make small changes). QA leader is also responsible for managing the whole test project. User with this role can create test plans, generate test reports or adjust schedules, etc.

Redmine is a tool (http://www.redmine.org) designed for project management. It comprises a centralized repository for managing requirements, version management and time tracking of various issues. Redmine repository comprises data on various issues, versions, projects, user roles, etc. It can be supplemented with extension fields, attachments, communication messages, etc. The whole repository is based on almost 60 tables. An important feature is the capability to extend the data model (overall repository model is fixed, however custom fields can be defined). Redmine repository is used for tracking requirements and their analysis. It is helpful to create work items and assign them to developers. Redmine includes tables for storing data changes describing related issues. This functionality is very useful for tracking and analyzing changes, comparison of revisions, etc. Redmine provides also the ability to create and store wiki pages.

The Redmine includes different type of timelines: Gantt charts, calendar, roadmap, deadlines, and other features that help keeping track of what's going and what is the status of the project. Redmine supports task assignments, bug-tracking and ticketing, allowing project managers to track progress of each feature, problem handling, and plan resources in advance. Redmine has also functionalities for various notifications (e.g. emails, RSS feeds) and document management. Like TestLink, Redmine can be configured to protect sensitive data.

Mantis Bug Tracker is a bug tracking system (https://www.mantisbt.org) which can serve also as a project management tool. It supports and integrates with many web based version management systems (e.g. SCM, GitHub, SourceForge), and admits integration of options (plug ins). Some mechanisms are available to visualize relations between various issues and prepare documentation (it includes change logs, audit trials, related to registered issues). They also provide multi-level access control, built-in search engines and report generation.

In Mantis Bug Tracker we can distinguish two repositories. The first one comprises tables with bug data. Information on bugs can be extended with custom fields, notes, attachments, etc. Some tables keep information on versioning, changes of data about bugs, relationships between bugs (e.g. common source of two bugs). The second repository relates to project configuration (including hierarchy, user roles, profiles and preferences). Additional technical tables relate to configuration data about plugins, email and other notification data.

TRAC is a bug tracking system (http://trac.edgewall.org) with project management features. It allows to track changes in issue descriptions, and also can help creating links (and integrate) between bugs, tasks, changes, related files. One of

functionalities of the TRAC is a timeline, that shows all current and past project events (gives an overview and tracks the project progress). TRAC provides a roadmap, which shows the plans ahead, lists the upcoming milestones, etc. It includes advanced hyperlinking options (to hyperlink information between bugs, revisions and wiki contents), fine-grained permissions options and customized reporting.

In TRAC the progress of individual bugs, requests, and other issues, are tracked using unique tickets (sequential numbers). Each detected problem receives a ticket. All changes for bug (ticket) are recorded and they can be viewed in the timeline for its status changes. There is also a simple way in TRAC, to connect overlapping tickets (where users report the same thing). TRAC has also extensive searching and filtering options for tickets by version, severity, owner, project component or priority. One of the unique things of TRAC is a timeline of each individual ticket. Project changes can be viewed in relevance to chronology of events (code changes highlighted). TRAC provides GUI for browsing and management version management tools (e.g. SVN, CVS, Git or other SCM systems). TRAC capabilities can also be extended with plugins.

We had a possibility of analysing several repositories collected during development and maintenance of some real projects. We have found that they comprise a lot of data and that the companies used them in a limited way. Hence, we decided to drill down the contents of these repositories to assess the value of the comprised information as well as to propose some measurement and evaluation schemes (discussed in section 4). As compared with published results in the literature we have observed the possibility of more detailed and fine-grained analysis.

4. Analyzing project repositories

We have analyzed many repositories of real projects from two companies. Here, we concentrate on 13 projects within this group projects P1-P11 were managed using custom tool similar to Redmine, however the history of state changes of problem reports have not been registered, project P12 used Mantis Bug Tracker, and P13 TestLink tool. In classical approaches authors use the notion of failure, bug or error, we have generalized this notion as problem. In particular, the registered problem after analysis may be rejected as falsely identified (e.g. due to incompetence of tester or user) or not important. Analyzing problem handling we distinguish user and tester perspectives and trace related handling processes which in practice may involve many intermediate states starting from registration, analysis, correction, validation, inclusion in the release, etc. In the literature problems (bugs) can be open or closed. In the analyzed projects we distinguish many reasons for closing problems, which better describe development and maintenance processes. Various statistics we have published in [6,7], in this section we give other complimentary statistics and interpretations. Moreover, we identify the lacking infor-

mation which could improve assessment of development and maintenance processes, the more that many tools (section 3) give such possibility, unfortunately neglected in practice.

Tools related to managing software testing (e.g. TestLink, compare section 3) provide the capability of tracing the progress of testing, e.g. plots of executed test scenarios, test cases in time in relevance to the assumed deadlines, correlation with system modules, correlation with testers, etc. An interesting issue is to identify the statistics of test distribution results. In general, we can distinguish passed (P - no problem identified), failed (F - incorrect behavior of the tested object) and blocked (B - test cannot be completed due to some problems e.g. lack of cooperating module) tests. This is illustrated in tab. 1 for project P13. Columns show the results for the specified weeks. The test preliminary phase relates to weeks 1-29 and covers small number of executed tests due to tester learning activities. Failed and blocked tests result in registration of the problem and initiating its handling, which is usually handled by other tools (e.g. Mantis BT, TRAC, section 3) and is discussed later on. Usually, resolution of blocked tests problems is simpler than the failed ones (e.g. code correction needed). Having collected such statistics on testing we can get some hints for predicting the number of problems in function of designed test cases. This prediction could be more precise if correlated with complexity of tested modules (e.g. lines of code, McCabe or Halstead measures [8,9]), unfortunately this is scarcely available information in repositories. Another issue is measuring engagement of testers (e.g. man hours), usually neglected in repositories.

Table 1. Distribution of test results in time for project P13

week	1	11	14	26	30	31	32	34	35	36
B	1	1	0	0	0	7	0	0	45	0
F	2	1	1	1	3	9	0	26	13	3
P	1	8	1	0	82	65	6	52	11	14
week	37	38	39	40	41	42	43	44	45	46
B	109	75	69	27	24	0	0	0	0	0
F	76	34	77	47	11	2	0	0	0	0
P	220	124	134	156	97	3	23	9	15	12

Analyzing problem handling processes we use PHG graphs (section 2). In the case of project P12 we had relatively rich information in the repository to trace problems in detail. The complete PHG graph comprised 26 states (nodes) and 280 edges. However, many states have been visited by a small number of problems, so they do not describe typical situations. Hence, we have introduced the capability of analyzing reduced graphs e.g. taking into account a specified number of related problems. In fig. 1 we give such graph assuming minimum 50 problems in a node (it is worth noting that the total number of registered problems was about 4000). The edges of the graph are labeled with percent of problems transferred to another node. Such graph visualizes typical problem handling paths. The developed tool

allows us to trace also processing times in each node, dominating paths, etc. The nodes of the graph relate to the following states: problem registration (S4), general analysis (S5), request for additional information (S6), problem withdrawing (S7), code correction (S8), fix prepared (S9), fix uploaded (S10), problem reopened (S11), transfer to test preparation (S12), technical analysis - it involves IT environment (S13), testing (S14), testing suspended (S15), problem completed (S16), rejection acceptance (S20). In graph of fig. 1 we have deleted states S1-S3 of initial analysis. Moreover, transitions (incoming and coming out) related to nodes with less than 50 problems are not included.

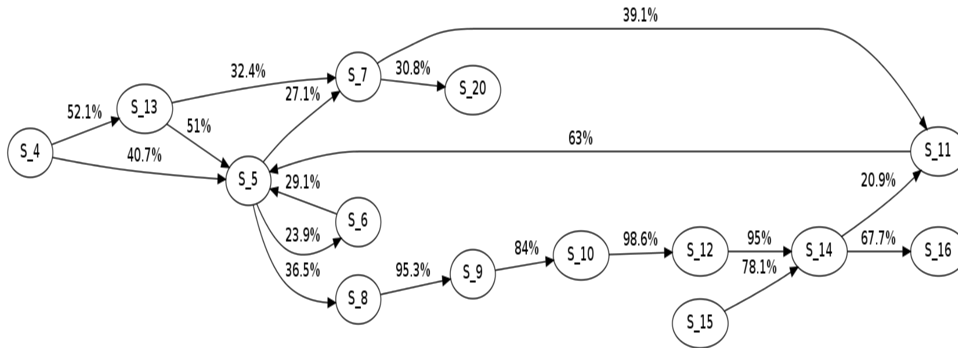

Figure 1. Reduced PHG graph for P12

An interesting issue is to identify and analyze the looping in problem handling, which is neglected in the literature, however it shows some problems in handling processes. In project P12 we have identified 1200 loops which involved from 2 to up 12 states. Each loop can be characterized by its structure (sequence of states) and saturation (number of circulating problems), for each problem we could also check the number of iterations in the loop. However, in most cases it is one. The loops with the highest saturation (shown in numbers) are as follows:

{S5|S7|S11|S5} – 403; {S5|S6|S5} - 306;
{S7|S11|S5|S7} – 258; {S13|S7|S11|S13} – 232;
{S5|S8|S9|S10|S12|S14|S11|S5} – 210;
{S14|S15|S14} – 206; {S11|S5|S7|S11} – 180;
{S13|S6|S13} – 162; {S13|S5|S13} – 155;
{S7|S11|S13|S7} – 134; {S14|S11|S5|S8|S9|S10|S12|S14} – 129;
{S12|S14|S11|S5|S8|S9|S10|S12} – 111; {S6|S5|S6} – 102;

It is worth noting that 881 loops involved only a single problem. There were 259 loops with 2-10 problems, 17 with 11-20 problems, 24 with 21-99 problems,

and 13 with more than 100 problems. In this last group the average circulation time was in the range 0.34-2.9 days. Minimal and maximal times were 0 and 65 days, respectively, Relatively higher average times occurred for loops with low saturation.

Another issue is comparing development and maintenance processes over many projects, or program modules. We had such possibility in relevance to projects P1-P11. As compared with project P12 the number of problem states was lower (8 states), moreover information on state changes was not available, however for each problem we had its appearance and completion time stamps, problem description, completion reason, etc. Despite these limitations we could derive some interesting features. In particular, we have compared the number of registered problems by testers (T) and users (U). The ratio U/T gives some measure on maintenance problems in relevance to test effectiveness during development. We illustrate this for selected projects (P3, P8, P10), subsequent numbers show U/T ratio for the specified modules (M_i) within the projects:

- Project P3: M_1 – 0.43, M_2 – 0.31, M_3 – 0.34, M_4 – 0.06, M_5 – 0.78, M_6 – 0.30, M_7 – 0.20
- Project P8: M_1 – 0.32, M_2 – 0.58, M_3 – 0.44, M_4 – 0.18, M_5 – 0.61, M_6 – 0.86, M_7 – 0.63, M_8 – 0.32, M_9 – 0.33, M_{10} – 2.23
- Project P10: M_1 – 0.15, M_2 – 0.96, M_3 – 0.27, M_4 – 0.52, M_5 – 0.86, M_6 – 0.12, M_7 – 0.33

In the presented statistics (profiles) the modules are ordered according to decreasing number of problems detected by testers (modules with lower number of problems are not included). In most projects the registered problem reports for the first modules dominate from 67 to over 90%. We do not present data for modules with less than 10 problems. It is worth noting that in P10 the second module (379 registered problems by testers) generates also many problems by the users (maintenance phase), the first module seems to be more reliable (689 tester problems versus 106 user problems).

The efficiency of handling problems can be visualized in a plot of open problems (i.e. remaining unresolved) in time. This is illustrated in fig. 2 for system P3 with time scale in months. An important issue is to correlate this plot with introduced revisions, they are shown as small rectangles in the upper part of the figure. Unfortunately, the related code complexity was not reported. In months 70-90 some increase of unresolved problems is observed, this queue has been handled quite effectively in subsequent few months. From month 91 the system shows good stabilization with a negligible number of open problems.

We can also look at the activities of testers and users in revealing problems. For an illustration in fig. 3 we give the distribution of problems detected by indi-

vidual testers and users ordered in a decreasing way. It is typical that some of them dominate (uneven distribution). Moreover, higher number of detected problems by testers than users confirms good testing effectiveness.

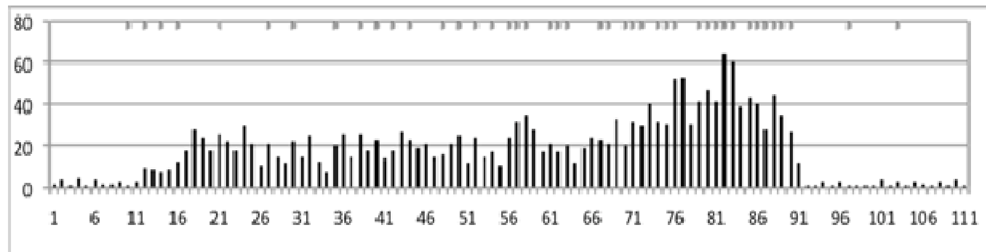

Figure 2. Distribution of open problems in time for project P3

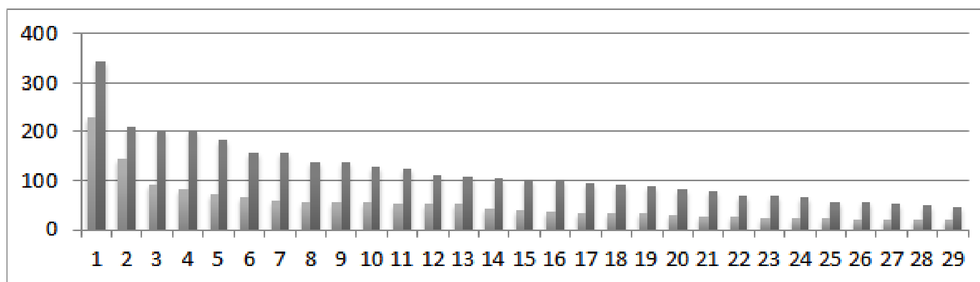

Figure 3. Distribution of revealed problems by 29 testers (black bars) and users (dashed bars) for project P3

Table 2. Distribution of specific problems in projects P1-P4

	DP		NR		NI		RE		RO	
	U	T	U	T	U	T	U	T	U	T
P1	7,0%	9,8%	3,3%	2,6%	0,0%	0,6%	0,3%	0.2%	4.5%	1.0%
P2	1,8%	8,1%	2.3%	1.8%	0.5%	0.4%	0.05%	0.2%	27.6%	8.2%
P3	5.5%	9.8%	2.4%	2.4%	0.1%	0.5%	1.2%	1.7%	0.1%	0.7%
P4	14.5%	9.7%	6.8%	2.8%	0.0%	0.4%	0.0%	0.4%	10.7%	4.8%

The completed resolved problems can be categorized according to ways (methods) of resolving them, e.g. correction included in the deployed version, correction waiting for deployment. It is worth noting that handling of many problems does not involve code corrections, so their costs can be low. In particular they relate to: DP - disqualified problems (falsely signaled, non-existing), NR – non reproducible problems (cannot be invoked for the described situation), NI – related to not important (or not used) functionality, RE – rejected due to excessive mitigation costs, RO – rejected due to other reasons. In tab. 2 we give their statistics in percent for projects P1-P4 in relevance to problems notified by users (U) and

tester (T). It is worth tracing the reasons of signaling DP, NR, NI and RO problems. Moreover, we have duplicated problems which need only identification and resolving only the representative instantiation. In the considered projects P1-P11 they contribute from a few up to 12 percent.

5. Conclusion

Having analyzed collected data in data repositories related to various projects (some of them are specified in [7]) we have identified that these data are helpful in managing and evaluating the quality of the project as well as development and service schemes. As compared with published reports in the literature we identify more useful information and propose more metrics. This allows us to identify deficiencies in problem handling processes and avoid them in new projects. On the other hand this analysis shows some shortage of collected data which results in the accuracy of modelling or result interpretation. In particular, the history of problem state changes was available only for one project, moreover information on the size and complexity of modules and code corrections was not available. The presented methodology is quite general and can be adapted to different development and maintenance schemes.

Further research is targeted at deriving characteristic features of developed code (e.g. various complexity measurers) and test coverage measures in order to include them in collected data of project repositories. This may facilitate finding more accurate models supporting project management.

REFERENCES

[1] Bluvband Z., Porotsky S., Talmor M. (2011) *Advanced models for software reliability prediction*, Proceedings of IEEE Annual Symp. on Reliability and Maintainability.

[2] Espinosa-Curiel I. E, Rodriguez-Jacobo J., Fernandez-Zepeda J. A. (2013) *A framework for evaluation and control of the factors that influence the software process improvement in small organizations*, Journal of Software Evolution and Process, 25, 393-406.

[3] Ferrer J., Chicano F., Alba E. (2013) *Estimating software testing complexity*, Information and Software Technology, 55, 2125-2139.

[4] Gupta A., Kapur R., Jha P.C. (2008) *Considering testing efficiency and testing resource consumption variations in estimating software reliability*, Int. Journal of Reliability, Quality and safety Eng., vol. 15, no. 2. 77-91

[5] Houston D. (2014) *A generalized duration forecasting model of test-and-fix cycles*, Journal of Software Evolution Process, 26, 877-889.

[6] Janczarek P., Sosnowski J. (2014) *Monitoring Software Development and Usage*, Przegląd Elektrotechniczny, Sigma NOT, vol. R. 90, no 2, 117-120.

[7] Janczarek P., Sosnowski J. (2015) *Investigating software testing and maintenance reports: Case study*, Information and Software Technology, vol. 58, 272–288.

[8] Kan S. H. (2003) *Metrics and Models in Software Quality Engineering*, Addison Wesley.

[9] Kozlov D., Koskinen J., Sakkinen M., Markula J. (2008) *Assessing maintainability change over multiple software releases*, Journal of Software Maintenance and Evolution, 20 (1), 31-58.

[10] Krini O., Borcsok J. (2012) *New scientific contributions to the prediction of the reliability of critical systems which base on imperfect debugging*, Proceedings of IEEE International Symposium on Telecommunications.

[11] Messquida A-L., Mas A. (2014) *A project management improving program according to ISO/IEC 29110 and PMBOK*, Journal of Software Evolution and Process, 846-854.

[12] Ogasawara H., Kusanagi T, Aizawa M. (2014) *Proposal and practice of software process improvement history since 2000*, Journal of Software Evolution and Process, 521-529.

[13] Petersen K. (2012) *A Palette of lean indicators to detect waste in software maintenance: A case study*, C. Wohlin (Ed.): XP 2012, LNBIP 111, Springer-Verlag Berlin, Heidelberg, 108–122.

[14] Pham H. (2006) *System Software Reliability*, Springer.

[15] Radjenovic D., Hericko M., Tprkar R., Zivkovic A. (2013) *Software fault prediction metrics, a systematic literature review*, Information and Software Technology, 55 1397-1438.

[16] Sideratos I. G., Platis A. N., Koutras V. P. Ampazis N. (2014) *Reliability analysis of a two-stage Goel-Okumoto and Yamada S-Shaped model*, W. Zamojski et al. (eds), DepCos-RELCOMEX, Advances in Intelligent Systems and Computing 286, Springer, pp. 393-402.

[17] Sommerville I. (2011) *Software engineering*, 9th edition, Pearson, Boston.

[18] Sosnowski J. (2006) *Testowanie i niezawodność systemów komputerowych*, Exit (in Polish).

[19] Sosnowski J., Sabak J. (2001) *Software reliability analysis in designing database oriented applications,* Proc. of the 27th Euromicro Conference, IEEE Comp. Society, 166-173.