Team Factors and Failure Processing Efficiency: An Exploratory Study of Closed and Open Source Software Development

Michael Grottke, Lars M. Karg
University of Erlangen-Nuremberg
Nürnberg, Germany
michael.grottke@wiso.uni-erlangen.de, lars.karg@acm.org

Arne Beckhaus
University of Freiburg
Freiburg, Germany
arne.beckhaus@is.uni-freiburg.de

***Abstract*—Researchers in the field of software engineering economics have associated team factors, such as team size and team experience, with productivity and quality. Since distributed and open source development have gained significance in the past few years, further empirical investigation is needed.**

Our study contributes to the empirical body of knowledge by addressing this development. In particular, we investigate the association between team factors and failure processing efficiency for closed source software releases of a large commercial software vendor and for open source software projects registered with SourceForge.net. We find significant links between team experience and the failure processing efficiency. However, our data does not show any evidence for adverse effects of distributed development. Our results further suggest that service level agreements and process governance are good tools to guarantee satisfactory processing times.

***Keywords*-closed source software, development teams, empirical analysis, failure processing, open source software**

I. Introduction

For decades, software development organizations have been confronted with the consequences of poor software solution quality and customer dissatisfaction [1]. Tremendous effort has been spent on improving software development to increase productivity and quality, while eliminating rework and waste [2]. Especially high release complexity and tight development schedules seem to cause the introduction of a large number of defects into the software [3]. However, many researchers claim that defects can never fully be avoided: Although software development is a structured process consisting of several stages, it is still human-centric [3]. It is thus advocated to keep rework effort low by aiming at finding and fixing defects as early as possible [4], and to derive process improvements based on the insights gained from the defects detected [5].

Since defects are not fully avoidable, efficient defect handling, which consists of recording, tracking, and solving of defects, is crucial for software development [6]. Thereby, one aspect software managers are confronted with is the improvement of failure processing efficiency to avoid the negative effects of long-pending failure reports. Several factors that may influence the failure processing efficiency have been investigated [7]–[9]. As software development remains a human-centric activity, team factors are of particular interest. Indeed, recent empirical studies have provided some evidence to the assumption that team factors affect the failure processing efficiency [9], [10].

Our research objective is to empirically investigate the influence of team factors on failure processing efficiency. We are especially interested in settings where no dedicated team for failure processing has been established and the processing is done by the development team itself. Though being uncommon in stages close to maintenance, such setups are frequently encountered in the implementation phase of large software development projects, especially when agile development practices are applied. A better understanding of the analyzed factors will help software managers direct process improvements as well as achieve a higher failure processing efficiency.

Because open source software development has recently gained in significance [11], we conduct our analyses in two research settings: at a commercial software vendor, and at SourceForge.net. Thus, our study does not only examine the influence of team factors on failure processing efficiency in the context of closed source software, but it also helps reveal differences between the open and closed source software development paradigms with regards to processing efficiency.

To the best of our knowledge, this is one of the first studies investigating team factors and failure processing efficiency in both closed and open source software development, relying on a rather large sample size and on large releases. It also differs from prior research, such as [12] and [13], since it studies the factors in a setting where no dedicated team for failure processing has been established.

The remainder of this paper is structured as follows: In Section II, prior research related to our study is presented. Next, Section III derives our conceptual research framework. In Section IV, we introduce the research sites as well as the variable measures, and we present the empirical results. In Section V, we discuss our findings, before presenting potential threats to validity in Section VI. The paper closes with Section VII, summing up the key findings and giving an outlook on future research.

II. Related Work

Several streams of research are devoted to failure processing in general and to failure processing efficiency in particular. These include research investigating bug trackers, such as Zhao and Elbaum [14] and Gupta and Singla [15], as well as work on failure processing effort prediction, such as Song et al. [16] and Grottke and Graf [17].

However, the research stream most closely related to our study is concerned with the task of failure processing itself, its efficiency, and the people involved. Frequently, the collaboration of the developers involved in processing an individual failure is investigated from a coordination-theoretical perspective [9], [18]. Different fault-, people-, and team-related factors have been investigated and linked to processing efficiency [8]. Studies report differences between closed and open source development projects and suggest that closed source development projects respond to failures more quickly than open source development projects [13]. This is in contradiction with Paulson et al. [19] and Raymond [20], who claim that a rapid response to reported failures is a particular strength of open source development. Other research studies team- and product-related factors in the context of maintenance. Most frequently, the influence of software complexity on the maintenance performance is studied; empirical evidence suggests that it is indeed a major factor [21].

Like many other researchers, Banker et al. [7] investigate how software complexity and functional size are linked with software maintenance and failure processing performance. They use data on 17 major applications, written in COBOL, from a major regional U.S. bank. Their study suggests that software maintenance performance is significantly affected by software complexity: As complexity increases, performance worsens. Based on this finding, they conclude that it is important to keep software complexity low during development to improve performance during the subsequent maintenance phase. This is an expected finding, since maintenance is conducted by a different team of developers, who are often unfamiliar with the software coding [7].

At Cisco Systems, Agrawal and Chari [12] investigate nine software products with more than 10,000 reported failures. With their study, they try to identify those factors having the highest influence on the repair times of the failure-related faults. Their results suggest that the repair times are influenced by several factors, such as fault severity, skill level, and tool support.

For post-release failures, Yu and Chen [13] compare the failure processing and fault correction efficiency in closed and open source development projects. They employ three metrics: the time it takes until a failure is reported after the software has been released, the time interval between the reporting of a failure and its assignment to a developer, and the time it takes to fix a fault after the related failure has been assigned to a developer. For their study, they rely on data of the NASA Ames projects, as well as three open source projects, namely, Apache Tomcat, Apache Ant, and Gnome Panel. Their results suggest that in closed source development projects failures are responded to more quickly, and the related faults are corrected faster.

Our study differs from prior research in several ways: First, we are interested in settings where no dedicated team for failure processing has been established and the processing is done by the development team itself. Prior studies, such as Banker et al. [7] and Agrawal and Chari [12], are devoted to settings where such a team is in place. Second, this is one of the first studies investigating team factors and failure processing efficiency in closed and open source software development. For instance, Agrawal and Chari [12] only focus on closed source software development projects, while Yu and Chen [13] merely propose efficiency metrics and compare them between closed and open source software development projects. Third and finally, our study is based on a rather large sample size and on large releases, while most prior research relies on a small number of small projects.

III. Research Framework

Our study of the influence of team factors on failure processing efficiency is based on the research framework shown in Figure 1. It investigates the following two research questions in settings where no dedicated team for failure processing has been established:

1) *Which team factors influence the average pending time of failures reported for a release?*
2) *Which team factors influence the average solving time of failures reported for a release?*

Consistent with prior research on failure processing [12], [13], we define efficiency in terms of the time spent on processing a failure. As depicted in Figure 2, we divide the overall processing time for an individual failure into two parts: the time it takes to assign a developer after a failure has been reported (in the following referred to as ‘pending time’), and the time it takes to close the failure after a developer has been assigned (hereafter referred to as ‘solving time’). Since all team factors studied are measured for an entire release, we cannot use them to model the processing time of an individual failure. Instead, we use the average pending and solving times of all failures related to a release as the dependent variables to be explained.

Due to the importance of human performance in software development, investigations of software quality have often considered team factors. In our exploratory study we make use of team factors identified by previous work.

For instance, Krishnan [10] examines the role of team factors for the quality of packaged software products at a leading software vendor. For his study he relies on a

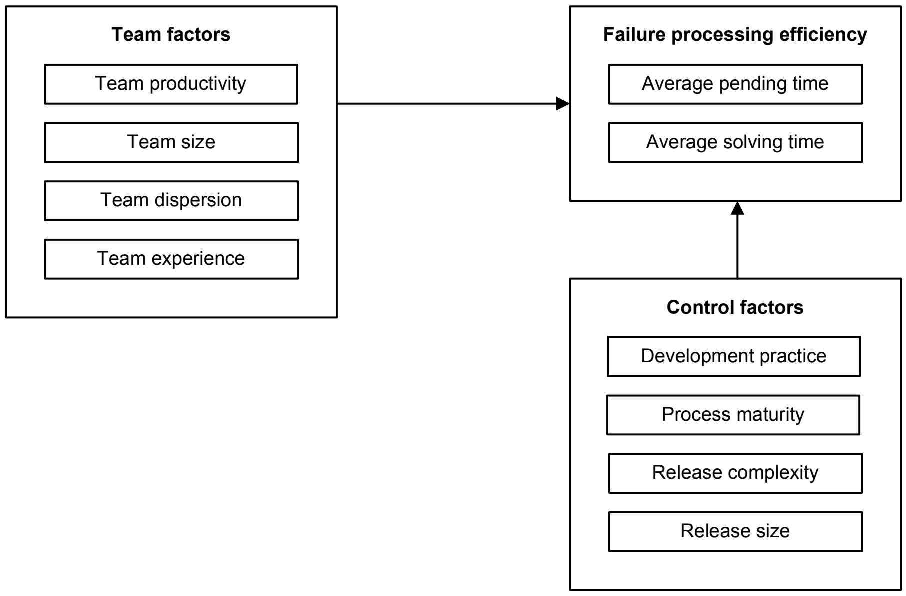


Figure 1. Conceptual research framework

sample of 37 products. His results suggest that increased team capabilities and team domain experience reduce the number of shipped product faults, whereas the experience of the team with the programming language used may not affect quality.

Bird et al. [22] study the effect of team dispersion on the product quality of Windows Vista in terms of its post-release failures in an intra-organizational setting. In contrast to prior research, they provide evidence that distributed development has little or no effect on the product quality of Vista. They conclude that for large software development projects, distributed development can work if the right development practices are chosen.

We focus on four team factors, expecting the following effects regarding failure processing efficiency (see Figure 1):

Team productivity: An increase in development productivity might adversely affect the average pending and solving times, because the developers may focus more strongly on generating new code than on coping with existing problems.

Team size: A larger team might increase the average pending time, since it becomes more difficult to identify the responsible developer who should process a specific failure report.

Team dispersion: A higher team dispersion might also adversely affect the average pending time, because identifying the developer responsible for a failure becomes more complicated.

Team experience: Better experience can be assumed to increase failure processing efficiency (in terms of average pending and solving times), since the team members get more accustomed to dealing with failures.

Prior research suggests that process- and product-related factors may influence software development performance. We should thus control for such factors.

For example, MacCormack et al. [23] show that employing a development practice particularly tailored to the type of software developed may improve quality. Moreover, the maturity level of the development process seems to affect the stability and reliability with which processes are performed [7], [24]. Furthermore, empirical research provides evidence that software release size and complexity have a significant effect on development performance [7]: As the software grows in size and/or complexity, it becomes more difficult for a developer to understand its dependencies. On the one hand, this increases the likelihood of introducing faults into the software. On the other hand, the time required to comprehend the cause of a software failure and to fix the related fault tends to get longer.

In our research framework we therefore control for the four process and product influences shown in Figure 1.

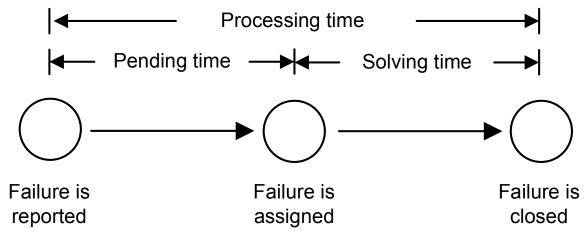


Figure 2. Failure processing steps

IV. Methods, Analysis, and Results

In this section, we first introduce the research sites, the variable measures, and the data analysis techniques used. We then present the empirical results of our study.

A. Research Sites and Data Collection

For our study, we gather data at two research sites to investigate the differences between closed source commercial software development and open source development. The first site is a commercial software vendor (hereafter referred to as the Vendor), where we cooperated with one development unit. This particular unit is in charge of new product development and does not have a dedicated team responsible for failure processing. Development is globally distributed, but it follows a strict development model.

The second research site is SourceForge.net. At the end of March 2009, the time we gathered the data for our study, SourceForge.net was hosting more than 140,000 open source software projects. In line with the open source philosophy, SourceForge.net makes all project data stored on its platform available to research. The large sampling population and the wealth of accessible data per project make SourceForge.net an excellent research site for studying open source software development [25]. By nature, open source development teams are virtually distributed. We can further assume that there are no dedicated teams for failure processing.

In the following, we discuss in detail the data sources used and the data collection methods employed at these two research sites.

1) Commercial Software Vendor: At the Vendor's site, we use data from three major data sources. The first one is the **time reporting and cost accounting system**, where all effort-related information is collected. Employees are required to record all times expended at the activity level on a daily basis. Since each activity is linked to one specific project, the effort and times per release can easily be aggregated. To ensure proper usage of the different activity types, data quality and validity, random samples of the reported effort data are cross-checked (e.g., by controlling experts). In addition to effort data, we also extract information on project staffing and organizational structure from the system.

Our second data source is the **bug tracker**, which we use for accessing failure data. During development, when a tester or quality expert experiences an unexpected software behavior, s/he enters a failure report into the bug tracker. Among other data, this report contains the time of its creation and information on the unexpected behavior. During failure processing, further information, like the closing date, is added.

The **software configuration management system** is used to gather information about the software release. Since software changes constantly, multiple versions of the same release, including source code and requirements, need to be managed. Through the software configuration management system, we access data on the size and the complexity of the software, as well as on the number of requirements implemented.

Due to service level agreements and process governance, data quality and reliability at the Vendor's site can be expected to be high. To ensure internal validity, it is recommended that the objects under investigation are homogeneous [7]. Since all releases were developed at the same unit within a short time span, the organizational structure, the maturity of the processes, and the development practices employed can be assumed to be identical for all releases. Expert interviews conducted at the Vendor's site confirm this conjecture. Hence, besides data completeness we only use one further criterion for release selection: We exclusively consider releases written in one specific programming language; those using other programming languages were not taken into account. Based on these two selection criteria, we compile a set of 31 releases.

2) SourceForge.net: Our study relies on information related to three SourceForge.net services. The source code of each considered project is accessed via the **source code repository**. Furthermore, we gather basic information on each reported failure (e.g., its processing time) from the **bug tracker**. Finally, the **general project statistics** are used to collect project-related information, such as team size.

Not all projects hosted at SourceForge.net are suitable for our study. As before, the projects examined should not be too heterogeneous; in addition, we also want them to be comparable to the releases investigated at the Vendor's site. Moreover, all required data has to be available; for example, the projects need to make active use of the bug tracker, allowing us to calculate processing times for failures of different priorities. Furthermore, the processes employed have to be mature enough for data collection to ensure reliable data. We therefore follow a strict selection process.

In a first step, we collect the first 250 projects listed in each of the two categories 'Enterprise' and 'Financial' in the SourceForge.net software map. These categories are non-exclusive. Dropping the 17 duplicates, we obtain a set of 483 unique projects.

In a second step, we then rank these projects by two criteria: the development activity (a metric calculated by SourceForge.net), as well as the number of downloads. Based on these criteria, we select the 354 top projects.

In a third step, we drop projects with a development status of less than 5 (raising doubts about the the process and product maturity), as well as projects for which less than 5 developers are registered. Moreover, we omit those projects for which manual checks reveal that the bug tracker is not actively used or that no priorities are assigned to the failures.

This three-step approach results in a final set of 31 projects. For our analysis, we include the latest release of each of these projects.

B. Variable Measurement

In the following, we describe how the factors are operationalized in our study. For each of the factors included in our models, the name of the variable is given. We also discuss which factors are excluded from our study, because they are constant, or unobserved, or assumed to be highly correlated with other factors.

1) Dependent Factors: For the dependent variables, we use the same metrics at both research sites.

Average pending time (AvgPendTime): We calculate the pending time of an individual failure as the time span (in minutes) between the initial failure report and the first action taken in response to it. For a release or project, the average pending time is computed as the arithmetic average of the pending times of all high-priority failures reported. We only take into account high-priority failures, because the failure processing efficiency shows best in the ability to quickly deal with urgent problems [8]. More specifically, we consider the failures with the highest priority at the Vendor's site, and the failures with priorities 7, 8, and 9 at SourceForge.net.

Average solving time (AvgSolvTime): We calculate the solving time of an individual failure as the time span (in minutes) between the first action taken in response to the failure report and its final closure. For a release or project, the average solving time is calculated as the arithmetic average of the solving times of all high-priority failures reported. The priorities considered are the same as for the average pending time.

2) Independent Factors: **Team productivity (TeamProd):** Development productivity until software shipment is measured as the software size in source lines of code (SLOC) divided by the development effort in person days. As is often the case for open source development projects [26], effort data is not available for our sample collected at SourceForge.net. Therefore, we can measure development productivity at the Vendor's site only.

Team size (TeamSize): The team size is given by the number of people involved in the software development project. For the Vendor, we measure this variable in terms of the number of developers staffed on the project developing the specific release. In the case of the open source software releases, we utilize the number of people involved in the project according to the SourceForge.net project statistics. This metric is derived from explicit registrations with a project and therefore represents the core team processing failures, while the periphery of occasional failure reporters is ignored [24].

Team dispersion (TeamDisp): For the Vendor, team dispersion is calculated from the fractions of total effort spent at each one of the n development locations/centers ($\text{Effort}_1, \ldots, \text{Effort}_n$) [24]:

$$\text{TeamDisp} = 100^2 - \sum_{i=1}^{n} (\text{Effort}_i)^2.$$

Note that all effort fractions Effort_i are expressed as percentage values between 0 and 100.

Information on the fractions of total effort spent at each development location/center is not available at SourceForge.net. We could compute the team dispersion metric by assuming that each core team member represents one virtual development center and that the total development effort is evenly distributed among them. However, such assumptions are overly simplistic and cannot be expected to yield a valid measure. We therefore exclude team dispersion from our analysis of the open source development projects.

Team experience: It has been shown that better team experience can strongly increase performance, because the people are more familiar with the tasks they perform [27]. In the context of failure processing, an obvious metric of team experience is the number of failure reports handled by the development team members prior to the current release. However, we can only apply this metric at the Vendor's site, because only there we can access the history of failure reports with which the team members have previously been involved **(TeamExp_V)**. To measure experience in the SourceForge.net setting, we can only rely on the age of the project, a metric suggested by Au et al. [28]. For our open source development projects, we thus measure experience in terms of the number of minutes since the project was registered with SourceForge.net **(TeamExp_SF)**.

3) Control Factors: **Release size (FuncSize):** At the Vendor's site, there is a strict and formally defined requirements engineering process. As a consequence, all requirements tend to be of similar granularity. We can thus measure size of the closed source releases in terms of the number of implemented requirements. The different projects from SourceForge.net, however, do not follow the same requirements engineering process. For them, the number of requirements is thus not a reliable metric of functional size. Since the projects also utilize different programming languages, we cannot simply measure functional size by SLOC either. Employing the table provided at [29], we therefore convert the SLOC measures obtained with the SLOCCount tool [30] into function points [31]. While SLOC can only be roughly mapped to function points, for the open source projects this is still the most reliable measure of functional size available to us.

Since prior studies have shown that many software complexity metrics are highly correlated with software size [32], [33], we exclude complexity from our study to avoid collinearity issues [34] in our models.

4) Constant or Unobserved Factors: As mentioned in Section IV-A1, there is evidence that process maturity and development practice are constant over all of the Vendor's releases. Therefore, we do not include these two factors in our models for the closed source projects.

Likewise, due to the strict selection process employed when choosing projects from SourceForge.net we can as-

Table I
OLS REGRESSION RESULTS FOR ALL LOG-LINEAR MODELS

	Vendor		SourceForge.net	
Variables	ln(AvgPendTime)	ln(AvgSolvTime)	ln(AvgPendTime)	ln(AvgSolvTime)
(Intercept)	4.2855**	7.6073***	4.3089	2.2537
	(1.2681)	(1.4066)	(5.4356)	(4.0014)
ln(TeamProd)	0.0165	0.1250		
	(0.0686)	(0.0760)		
ln(TeamSize)	0.1011	−0.0040	0.2146	0.1836
	(0.1029)	(0.1141)	(0.1904)	(0.1415)
ln(TeamDisp)	0.0212	−0.0905		
	(0.1161)	(0.1287)		
ln($\text{TeamExp}_{\text{V}}$)	−0.0598**	−0.0531*		
	(0.0174)	(0.0193)		
ln($\text{TeamExp}_{\text{SF}}$)			0.6192	1.1572*
			(0.5797)	(0.4418)
ln(FuncSize)	0.4809**	0.3978*	0.0253	−0.0581
	(0.1677)	(0.1860)	(0.2430)	(0.1767)
$\mathcal{R}^2$	0.4364	0.3457	0.0991	0.3101
F-statistic	3.7170	2.5362	0.9536	3.8951
p-value	0.0124	0.0559	0.4293	0.0201
Sample size	30	30	30	30

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$. (Standard errors are given in parentheses.)

sume that the process maturity is at a comparable level across these open source projects. Information on the development practice is not available from SourceForge.net. Moreover, our discussion of the independent factors in the last section has shown that team productivity and team dispersion cannot be measured reliably for these open source projects. Consequently, we omit the factors process maturity, development practice, team productivity, and team dispersion in the models for the SourceForge.net releases.

C. Data Analysis Techniques

We conduct all our quantitative analyses with the statistical software package R [35].

Prior empirical research on software engineering economics recommends a log-linear model of project and release performance [7], [24]. We therefore apply a log transformation to all variables.

To test the adequacy of our models, we use Ramsey's RESET [36]; no misspecification is detected. The parameters of the log-linear models are then estimated via the ordinary least squares (OLS) approach. Note that we estimate the models for both research sites separately, because different variables are included and different metrics are employed.

To ensure that the assumptions of the OLS approach are met, we use a variety of specification checks for the estimated models. Both visual examination of the residual plots as well as the DFFIT measure [37] indicate one influential observation in the data of all four models. Each of the models is therefore re-estimated after removing the respective influential observation from the data set. The resulting parameter estimates are shown in Table I. They are very similar to the original ones.

Based on the reduced data sets, we use variance inflation analysis [38] to check for the presence of multicollinearity. For all models, the values of the variation inflation factors (VIF) range from 1.15 to 1.99, with a mean VIF of 1.47. Since potential problems due to multicollinearity are indicated by VIF values above 5.3 [39], we conclude that multicollinearity is not a serious issue in our analysis.

At the 5% significance level the Shapiro-Wilk test [40] cannot reject the normality assumption for the residuals of any of the four models. Furthermore, neither the Breusch-Pagan test [41] nor the White test [42] indicates any violation of the homoscedasticity assumption for any of the models. In other words, *there is no evidence that any of the assumptions of the chosen parametric data analysis technique are violated*.

D. Regression Results

For the **Vendor** models, the parameter estimates in Table I indicate that better team experience is associated with a shorter average pending time of failures, whereas a larger functional size tends to be linked to a longer average pending time. Similar results are obtained with respect to the average solving time. Since the regression coefficients in log-linear models represent elasticities, a 1% increase in team experience tends to reduce average pending time and average solving time by about 0.06% and 0.05%, respectively. However, a 1% increase in functional size is linked with a 0.48% increase in average pending time and a 0.39% increase in average solving time. No significant association between the remaining independent variables and the dependent variables can be established.

As for the analysis of the **SourceForge.net** project data, the low coefficient of determination $\mathcal{R}^2$ indicates that the model for the average pending time has hardly any explanatory power. Moreover, the high p-value associated with the small value of the F-statistic shows that the hypothesis that all regression coefficients are equal to zero cannot be rejected at any reasonable significance level, such as 5% or 10%. For the other model, our results suggest that better team experience is linked with longer average solving times; a 1% increase in team experience is associated with about a 1.16% increase in average solving time.

V. Discussion

The results for the **Vendor** indicate that larger functional size is associated with a lower failure processing efficiency. In fact, prior studies have shown that software size has a strong negative influence on efficiency in general [7], [24]. This finding can be explained by the fact that the functional size of a piece of software is related to its complexity. When a software product's source code grows in size, it tends to become more complex and more difficult to understand for its developers [7]. Hence, not only does it take longer for the developer to decide if s/he is responsible for a failure and should process it, but s/he also needs more time for solving the failure.

The positive association between team experience and failure processing efficiency observed at the Vendor's site is expected and consistent with prior research [24]. In general, it can be assumed that team experience and knowledge regarding a particular task increase as the team members repeat it over and over again. By processing more failures the developers become more efficient for two reasons: First, they get better in deciding who is responsible for a particular software failure; this effect improves the average pending time of failures. Second, the developers get better in processing the failures related to their coding; this effect reduces the average solving time. In consequence, failure processing efficiency increases.

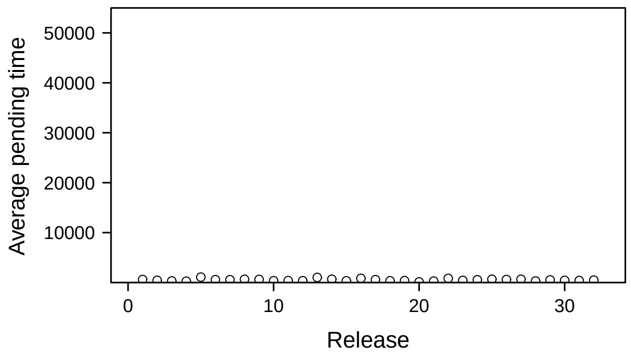
For the remaining factors, our study does not indicate any significant influences on the failure processing efficiency at the Vendor's site. As discussed above, the development practice employed could be the reason why team size and dispersion do not show any effect. This development practice might also ensure that the development team has enough time to process failures, regardless of the development productivity. Appropriate development practices may thus help guarantee efficient failure processing [22].

At **SourceForge.net**, the observed association between team experience and the average solving time is unexpected. We would have assumed that better experience results in a higher processing efficiency. However, remember that we followed Au et al. [28] in utilizing project age as the measure of team experience. In many open source projects developers join and leave frequently [43]. Over time, the improvements in experience on the part of the team members are therefore limited. On the contrary, in long-running projects developers tend to be faced with the challenge of processing failures related to parts of the code that had been implemented before they joined the development team.

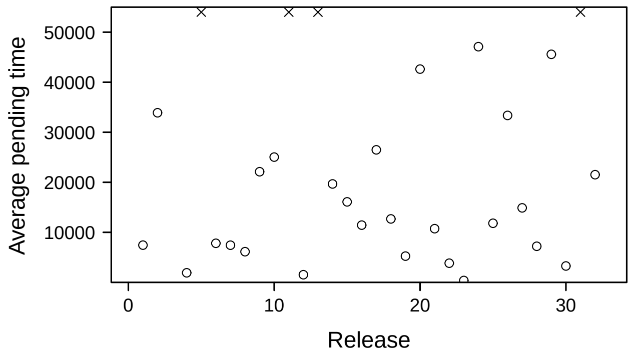
The factors release size and team size do not seem to have any effect on the failure processing efficiency. This may be explained by the fact that open source development projects are highly dispersed and driven by individual contributors. Therefore, they are modularized in a more fine granular way than closed source development projects. They can hence avoid the pitfalls of complexity at the expense of less team work and the boost in creativity that it implies.

According to Table I, the explanatory power of the SourceForge.net models in terms of the coefficient of determination $\mathcal{R}^2$ is lower than the one of the corresponding Vendor models. For the model of average pending time, it has not even been possible to reject the global hypothesis that all regression coefficients are jointly equal to zero (see Section IV-D). Figure 3 gives us some interesting insights into the differences between the closed and open source development projects with respect to the dependent variables. Obviously, for the open source releases the average pending and solving times show a much higher variation than for the releases of the commercial software vendor. (Note that the releases with average pending or solving times exceeding 53,000 minutes and thus falling outside the scale of the y-axes are shown as an 'x' at the top of the diagram.) Only parts of the huge fluctuations observed for the open source releases can be explained by the log-linear models. The lower variability for the closed source releases may be due to the service level agreements and process governance that are in place at the Vendor's site, ensuring reasonable failure processing times.

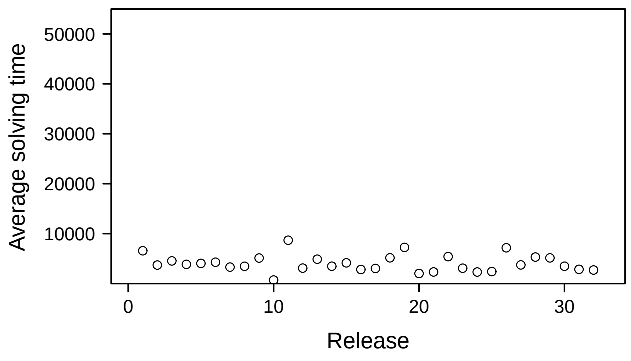
In the literature, the quick response to and solving of reported failures have been seen as a particular strength of open source development [20]. However, some researchers have provided evidence that closed source software development responds more quickly to failure reports [13]. Figure 3 gives an explanation for these contradictory findings. Based on a large sample of projects or releases, the overall failure processing efficiency tends to be higher in closed source development than in open source development, provided that service level agreements and process governance are in place. However, some individual open source projects



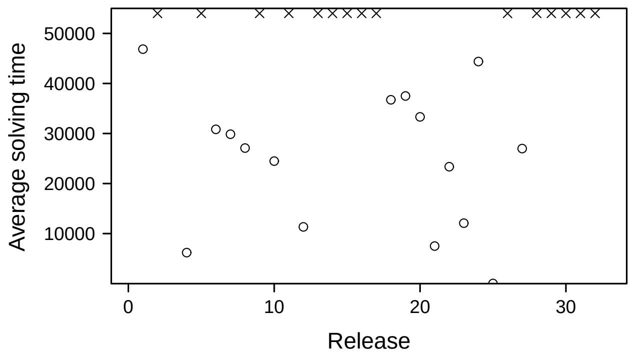
(a) Average pending time at the Vendor's site



(b) Average pending time at SourceForge.net



(c) Average solving time at the Vendor's site



(d) Average solving time at SourceForge.net

Figure 3. Failure processing efficiency

do indeed attain failure processing efficiencies comparable to those of closed source projects. Well-established open source projects are known for their large and very active community, which is helpful for fast failure processing. The results of studies focusing on single cases thus highly depend on the particular releases/projects chosen. For instance, Raymond [20] focuses on well-established large open source development projects, such as Linux or Gcc, whereas Yu and Chen [13] and our study consider smaller and/or less established ones.

In summary, service level agreements and process governance can be seen as good tools to guarantee satisfactory failure processing times. However, if the power of the community can be leveraged processing efficiency may be increased even further. In addition, it seems that the failure processing efficiency is not influenced as strongly by team factors as one might expect. Besides team factors, it can be assumed that failure-related factors (such as severity of the failure's consequences and clarity of the failure description) and the extent of tool support are crucial for the failure processing efficiency attained; see Crowston and Scozzi [9] and Gokhale and Mullen [8].

VI. Threats to Validity

Construct Validity: For the collection of our data, we relied on automated production-level quality tools, as well as on a strict selection process. We further excluded those factors from our study for which we could not guarantee reliable measurement. Therefore, we do not think that our data has been subject to any large measurement errors.

Internal Validity: Since our regression diagnostics showed no violation of the regression assumptions, we have reason to believe that the we correctly identified significant associations. Furthermore, we derived the factors from general work on software engineering performance; we therefore assume that the conclusions drawn are valid.

External Validity: Software development processes and projects do vary across research settings. It must therefore be assumed that our results are only valid in research settings comparable to ours. For instance, they probably cannot be transferred to embedded software development. However, our study might indicate important general principles and associations.

VII. Conclusions

In this study, we investigated the influence of team factors on failure processing efficiency. Since open source development has recently gained significance, we conducted our study in two research settings: at a large software vendor, and at SourceForge.net. We thus contribute to the empirical body of knowledge on closed and open source software development. Prior studies either focused on single releases or on closed or open source software development. As far as we know, this is one of the first studies investigating team factors and failure processing efficiency in closed and open source software development, comparing their failure processing efficiency, and relying on a rather large sample size and on large releases instead of small projects.

Our results indicate that appropriate development practices can prevent the negative effects of distributed development on the failure processing efficiency. Our study further suggests that the level of team experience affects failure processing efficiency. After processing many failure reports, team members know who is responsible for what part of the coding, and they are also able to solve the problems more quickly. Software managers should thus support learning in their organization to improve processing efficiency. Moreover, if service level agreements and process governance are in place, closed source development teams seem to show consistent failure processing efficiencies. However, well-established open source development projects can achieve comparable results.

Further research should especially be devoted to distributed development and the concepts of team experience and learning. Since recent studies have suggested that the adverse effects of distributed development can be avoided by adequate development practices, more research is needed to understand what aspects of such practices are of importance. As team experience has been shown to have a significant positive association with failure processing efficiency in closed source projects, it would be interesting to investigate how learning takes place in software development in general.

References

[1] J. A. Whittaker and J. M. Voas, "50 years of software: Key principles for quality," *IT Prof.*, vol. 4, no. 6, pp. 28–35, 2002.

[2] P. Middleton and J. Sutton, *Lean Software Strategies: Proven Techniques for Managers and Developers*. New York: Productivity Press, 2005.

[3] W. S. Humphrey, "The software quality challenge," *CrossTalk, The J. Def. Softw. Eng.*, vol. 21, no. 6, pp. 4–9, 2008.

[4] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 2001.

[5] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar, and R. Chillarege, "A case study of software process improvement during development," *IEEE Trans. Softw. Eng.*, vol. 19, no. 12, pp. 1157–1170, 1993.

[6] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River: Prentice Hall PTR, 2002.

[7] R. D. Banker, G. B. Davis, and S. A. Slaughter, "Software development practices, software complexity, and software maintenance performance: A field study," *Manage. Sci.*, vol. 44, no. 4, pp. 433–450, 1998.

[8] S. S. Gokhale and R. Mullen, "Software defect repair times: a multiplicative model," in *Proc. 4th International Workshop on Predictor Models in Software Engineering*, 2008, pp. 93–100.

[9] K. Crowston and B. Scozzi, "Bug fixing practices within free/libre open source software development teams," *Journal of Database Management*, vol. 19, no. 2, pp. 1–30, 2008.

[10] M. S. Krishnan, "The role of team factors in software costs and quality," *Information Technology & People*, vol. 11, no. 1, pp. 20–35, 1998.

[11] G. von Krogh and E. von Hippel, "The promise of research on open source software," *Manage. Sci.*, vol. 52, no. 7, pp. 975–983, 2006.

[12] M. Agrawal and K. Chari, "Software effort, quality, and cycle time: A study of CMM level 5 projects," *IEEE Trans. Softw. Eng.*, vol. 33, no. 3, pp. 145–156, 2007.

[13] L. Yu and K. Chen, "Evaluating the post-delivery fault reporting and correction process in closed-source and open-source software," in *Proc. 5th International Workshop on Software Quality*, 2007, p. 8.

[14] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *J. Syst. Softw.*, vol. 66, no. 1, pp. 65–75, 2003.

[15] A. Gupta and R. K. Singla, "An empirical investigation of defect management in free/open source software projects," in *Advances in Computer and Information Sciences and Engineering*, T. Sobh, Ed. Dordrecht: Springer Netherlands, 2008, pp. 68–73.

[16] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 69–82, 2006.

[17] M. Grottke and C. Graf, "Modeling and predicting software failure costs," in *Proc. 33rd Annual IEEE International Computer Software and Applications Conference*, 2009, pp. 180–189.

[18] K. Crowston, "A coordination theory approach to organizational process design," *Org. Sci.*, vol. 8, no. 2, pp. 157–175, 1997.

[19] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 246–256, 2004.

[20] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol: O'Reilly, 2001.

[21] C. F. Kemerer, “Software complexity and software maintenance: A survey of empirical research,” *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, 1995.

[22] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, “Does distributed development affect software quality? An empirical case study of Windows Vista,” in *Proc. 31st International Conference on Software Engineering*, 2009, pp. 518–528.

[23] A. MacCormack, C. F. Kemerer, M. Cusumano, and B. Crandall, “Trade-offs between productivity and quality in selecting software development practices,” *IEEE Softw.*, vol. 20, no. 5, pp. 78–85, 2003.

[24] N. Ramasubbu, S. Mithas, M. S. Krishnan, and C. F. Kemerer, “Work dispersion, process-based learning, and offshore software development performance,” *MIS Q.*, vol. 32, no. 2, pp. 437–458, 2008.

[25] J. Howison and K. Crowston, “The perils and pitfalls of mining SourceForge,” in *Proc. International Workshop on Mining Software Repositories*, 2004, pp. 7–11.

[26] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” in *Proc. 4th International Workshop on Mining Software Repositories*, 2007, p. 1.

[27] R. S. Huckman, B. R. Staats, and D. M. Upton, “Team familiarity, role experience, and performance: Evidence from Indian software services,” *Manage. Sci.*, vol. 55, no. 1, pp. 85–100, 2009.

[28] Y. A. Au, D. Carpenter, X. Chen, and J. G. Clark, “Virtual organizational learning in open source software development projects,” *Inf. Manage.*, vol. 46, no. 1, pp. 9–15, 2009.

[29] QSM, “Function point programming languages table, Version 3.0,” 2005, online: http://www.qsm.com/FPGearing.html.

[30] D. A. Wheeler, “SLOCCount,” online: http://www.dwheeler.com/sloccount/.

[31] A. J. Albrecht and J. E. Gaffney, “Software function, source lines of code, and development effort prediction: A software science validation,” *IEEE Trans. Softw. Eng.*, vol. 9, no. 6, pp. 639–648, 1983.

[32] B. W. Boehm, *Software Engineering Economics*. Upper Saddle River: Prentice Hall PTR, 1981.

[33] J. C. Munson and T. G. Khoshgoftaar, “Software metrics for reliability assessment,” in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 493–529.

[34] W. H. Greene, *Econometric Analysis*, 6th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2008.

[35] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2010, ISBN 3-900051-07-0. Online: http://www.R-project.org.

[36] J. B. Ramsey, “Tests for specification errors in classical linear least squares regression analysis,” *J. Roy. Statist. Soc. B.*, vol. 31, no. 2, pp. 350–371, 1969.

[37] D. A. Belsley, E. Kuh, and R. E. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. New York: John Wiley & Sons, 1980.

[38] D. W. Marquardt, “Generalized inverses, ridge regression, biased linear estimation, and nonlinear estimation,” *Technometrics*, vol. 12, no. 3, pp. 591–612, 1970.

[39] J. Hair, R. Anderson, and B. Babin, *Multivariate Data Analysis*. Uppder Saddle River: Pearson Prentice Hall, 2006.

[40] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[41] T. S. Breusch and A. R. Pagan, “Simple test for heteroscedasticity and random coefficient variation,” *Econometrica*, vol. 47, no. 5, pp. 1287–1294, 1979.

[42] H. White, “A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity,” *Econometrica*, vol. 48, no. 4, pp. 817–838, 1980.

[43] J. Bitzer and P. J. Schroder, *The Economics of Open Source Software Development*. Wakefield: Emerald Group, 2006.