Investigating the Impact of a Measurement Program on Software Quality

Houari Sahraoui[a], Lionel C. Briand[b], Yann-Gaël Guéhéneuc[c], Olivier Beaurepaire[d]

[a] *DIRO, Université de Montréal, Canada, sahraoui@iro.umontreal.ca*
[b] *Simula Research Laboratory & University of Oslo, Norway, briand@simula.no*
[c] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada,*
yann-gael.gueheneuc@polymtl.ca
[d] *Rail Solutions, SNCF, France, olivier.beaurepaire@sncf.fr*

Abstract

Context: Measurement programs have been around for several decades but have been often misused or misunderstood by managers and developers. This misunderstanding prevented their adoption despite their many advantages.

Objective: In this paper, we present the results of an empirical study on the impact of a measurement program, MQL ("Mise en Qualité du Logiciel", French for "Quality Software Development"), in an industrial context.

Method: We analyzed data collected on 44 industrial systems of different sizes: 22 systems were developed using MQL while the other 22 used ad-hoc approaches to assess and control quality (control group, referred to as "ad-hoc systems"). We studied the impact of MQL on a set of nine variables: six quality factors (maintainability, evolvability, reusability, robustness, testability, and architecture quality), corrective maintenance effort, code complexity, and the presence of comments.

Results: Our results show that MQL had a clear positive impact on all the studied indicators. This impact is statistically significant for all the indicators but corrective maintenance effort.

Conclusion: We bring concrete evidence that a measurement program can have a significant, positive impact on the quality of software systems if combined with appropriate decision making procedures and corrective actions.

Key words: Measurement Program, Software Quality, Empirical Study

1. Introduction

The software engineering community generally agrees on the usefulness of measurement programs and quality-aware processes to improve the quality of software systems. This consensus has evolved from the pioneering work by DeMarco [1] and Fenton [2] to current trends in metrics definitions [3] and capability and maturity models, for example [4]. Yet, only a few empirical studies [5, 6, 7, 8, 9, 10] in industrial contexts support this consensus.

In this paper, we present a study to assess the impact of a measurement program in an industrial context. This large-scale controlled study targeted a measurement program at SNCF, the French public railway company [11]. The results obtained show that a well-defined measurement program helps significantly improve the quality of the produced software systems and, thus, provides empirical evidence that the implementation of such a program can be cost-effective in industry.
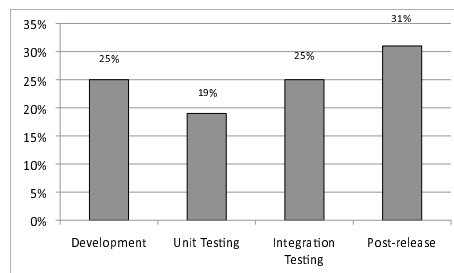

Figure 1: Defect Distribution in DSIV Projects.

The implementation of the measurement program at SNCF and the impact study presented in this paper were motivated by economical reasons. Indeed, SNCF is a key carrier of freight and passengers in France and Europe. After the deregulation of the transportation sector in Europe in the nineties and in anticipation of the opening of the sector to the European competition in 2003, SNCF faced many challenges for cost reduction. In particular, DSIV,

2

SNCF IT division, was the first entity to control its costs after two internal studies conducted in 2000 and 2001 revealed that it was necessary to improve the internal software development process and the management of sub-contracted projects. These studies found that more than 30% of the faults were detected after release, as shown in Figure 1. Following these studies, DSIV managers decided to define and implement a measurement program to reduce the costs and improve the quality of the delivered systems.

At the end of 2003, the MQL measurement program ("Mise en Qualité du Logiciel", French for "Quality Software Development") was implemented in DSIV. Before the implementation of MQL, no standard measurement program was uniformly used across project teams. Few teams followed repeatable development/maintenance processes. The quality team had to rely on testing and code inspection (check lists) to assess the quality of software systems and to make decisions. MQL is now used internally and by subcontractors during software development and maintenance. It is partly based upon the ISO 9126 Standard [12] and was developed using the Goal-Question-Metric (GQM) approach [13]. After two years of operation of the MQL measurement program, we conducted an empirical study to evaluate the impact of MQL on the quality of the delivered systems. We selected 22 systems developed/maintained using MQL for quality control. As control group, we selected 22 other similar systems for which ad-hoc approaches were used to assess and control quality (referred to as "ad-hoc systems"). We chose nine quality indicators, including six quality factors, one factor related to corrective-maintenance effort, and two code cognitive complexity factors. This study shows that MQL has a positive impact on all the indicators. It provides concrete, industrial evidence that a measurement program improves the quality of software systems.

The remainder of this paper is organized as follows. The following section presents the MQL measurement program. Section 3 describes the design of the MQL impact study. Section 4 presents and discusses the study results. Section 5 describes potential threats to the validity of our study and explains how we addressed them. Finally, Section 7 concludes and outlines future work.

3

2. The MQL Measurement Program

MQL was defined and implemented by DSIV, the SNCF IT department. DSIV is responsible for the development and maintenance of all software systems for SNCF customers. It is similar to large software companies with more than 1,000 employees, has an annual turnover of 200 million euros, and owns several hundred live systems. Systems use various technologies, from assembly language on mainframes to Java on J2EE platforms. DSIV works with several sub-contractors who develop and maintain many of the systems.

MQL was gradually implemented with the involvement of project managers and developers. Dozens of internal and subcontracted projects are now managed using MQL. The long-term objectives of MQL were to:

- improve predefined quality factors;

- reduce the cost of corrective maintenance;

- reduce code cognitive complexity to facilitate inspections and evolution;

We now present the process and the measures defined in the MQL measurement program. We also briefly discuss some lessons learned from the implementation of MQL.

2.1. MQL Process

Figure 2 shows the process of applying MQL during software development. Before release, project managers submit development artifacts (source code, documentation, test suites, etc.) for evaluation to the DSIV quality team. The evaluation takes the form of a quality analysis (the details are given in Section 2.2). After the evaluation, the results are discussed. In the case where quality anomalies are found (entities with abnormal measures), project managers could acknowledge these anomalies or explain why they could be justified. Justifications include performance consideration, schedule and cost constraints, and resource availability. Unjustified new anomalies trigger the creation of new
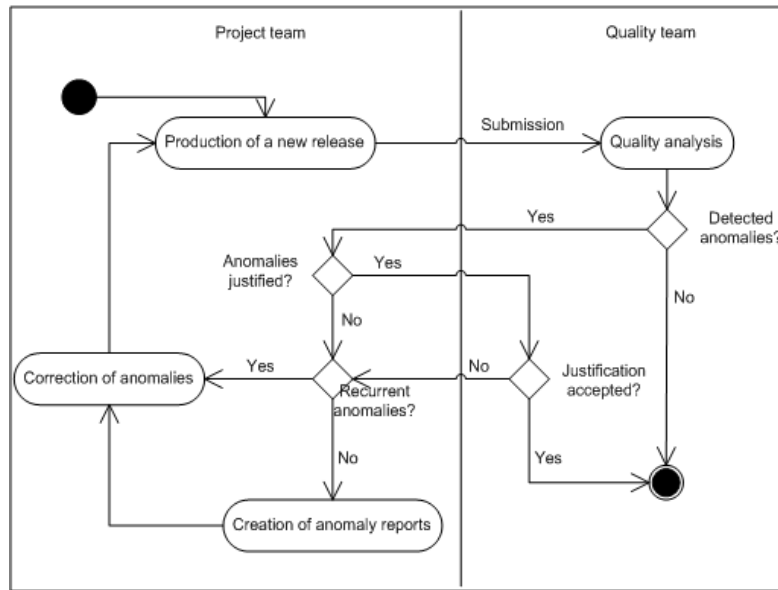
Figure 2: MQL Process (UML Activity diagram).

anomaly reports. New and recurrent anomalies are corrected prior to the next evaluation iteration.

For discussion and follow-up purposes, MQL evaluation results are published using an automated reporting mechanism on an internal Web portal.

Reports show the results of quality factors at different levels of granularities (methods, classes, packages/modules, systems). In addition to the factors, they include the occurrences of detected code and design smells. Figure 3 shows a screen shot of the first page of an analysis report, which presents the aggregated scores of various factors for a system. Detailed scores for basic elements (such as classes and methods) can be viewed using the navigation bar on the top of the page.

These results are also compared to those of previous releases, stored in a historical database. The Web portal helps to visualize the evolution of quality from release to release, which makes it possible to assess whether the objectives are reached and whether the project quality is improving. Figure 4 shows a screen shot of a page with the evolution of the quality factor scores for the last
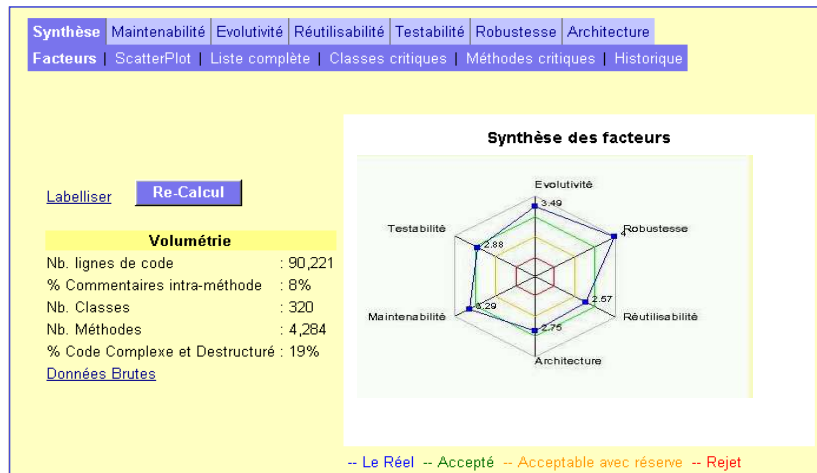
5

Figure 3: An Example of a Synthesis Page of the Web Portal[1].

10 releases of a system. After a steady period (versions 3 to 7), most the factors were improved in the last three versions.

The Web portal, as well as the provided information, requires proper authentication for access. Thus, managers only access the top-level aggregations of the data related to their projects, while developers have access to the detailed scores given to each class/file and method/function on which they are working. Quality experts have access to all the projects. The authentication mechanism ensures confidentiality of the information across projects and also reassures developers and managers that the Web portal, and by extension the measurement program, are not used for dubious purposes, such as hidden evaluations of personnel.

Using the Web portal and in cooperation with the project teams, the quality

---

[1]Translation from French. Tab-menu first line: *Summary* and the six factors *Maintainability, Evolvability, Reusability, Robustness, Architecture*. Tab-menu second line: *Factors, ScatterPlot, Complete list, Critical classes, Critical Methods, History*. Summary of size measures on the left: *Number of lines of code, % of intra-method comments, Number of classes, Number of Methods, % of complex and destructured code, Raw data*. Keviat diagram on the right: *Summary of the Factors*, and the six factors of the tab-menu second line. Color legend at the bottom: *Actual, Accepted, Acceptable with reservation, Rejected*

team defines improvement objectives and decides which actions to take based on measurement results, contextual information, and the history of the system. These actions include corrective and perfective maintenance, process improvement, and personnel training. Project teams are required to address the improvement objectives and perform agreed-upon actions for the subsequent releases.
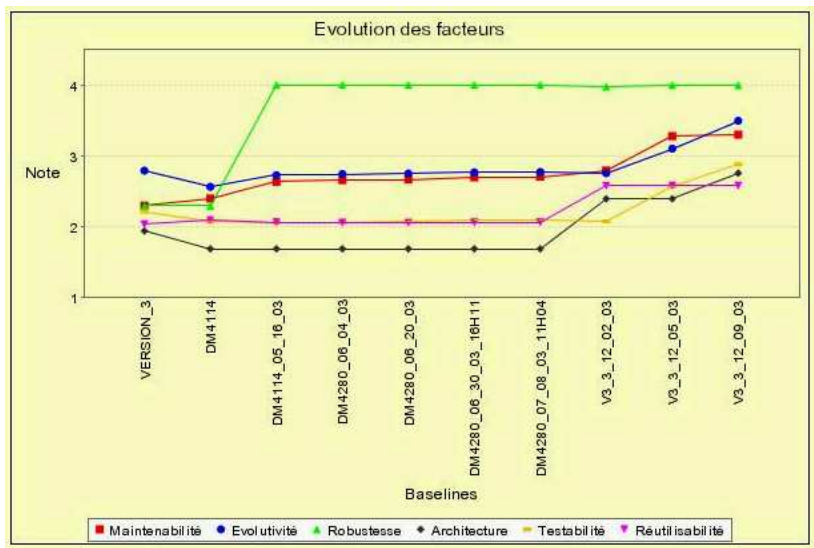

Figure 4: An Example of a Score Evolution Page of the Web Portal [2].

2.2. MQL Analyses

Within the MQL measurement program, five types of analyses are performed. All of them use/produce measurement and are then components of the measurement program. In the remainder of this section, we briefly present each of them.

2.2.1. Terminology

Before going further, we define the terminology that is used within MQL.

---

[2]Translation from French. Header: *Evolution of factors.* Plot Y-axis: *Score.* Color legend of curves: the six factors in Figure 3.

A quality **Factor** is a quality characteristic (according to the definition of the ISO 9126 Standard [12]) that might impact the cost of development and maintenance activities. In MQL, six factors were defined and used: maintainability, evolvability, reusability, robustness, testability, and architecture quality. Quality factors cannot be directly measured on the software products. They are estimated based on a set of criteria.

A **criterion** is a product internal attribute. It could be used to estimate one or more factors. There are four types of criteria: (1) structural attributes such as size and dependencies between elements [14], (2) the presence of anti-patterns such as *copy-paste* [15], and (3) the conformance to standards and conventions such as naming conventions [16]. There is also one additional criterion that is not an internal attribute, namely unit-test coverage. Criteria are evaluated using **metrics**.

The relationships between factors and criteria and between criteria and metrics are defined following a GQM approach. In the reminder of this section, we summarize how the process of evaluating factors using criteria, and criteria using metrics. The detailed application of GQM to the construction of the measurement program MQL is beyond the scope of this paper.

2.2.2. From Metrics to Criteria

Basic metrics are extracted from the code using commercial tools to calculate the scores of the criteria. Each criteria is evaluated on a 1-to-4 scale, where 1 means unacceptable and 4, acceptable. The mapping of the metric values to the 1-to-4 scale is realized by sets of satisfaction-level rules. For example, the criterion complexity of a method, a structural attribute, is derived from the metric $v(g)$, McCabe's cyclomatic complexity [17] using the following rules:

$$v(g) \geq 15, Complexiy\_score = 1$$
$$10 \leq v(g) < 15, Complexiy\_score = 2$$
$$5 \leq v(g) < 10, Complexiy\_score = 3$$
$$v(g) < 5, Complexiy\_score = 4$$

Satisfaction-level rules and thresholds are defined following the company standards, internal experiments, and literature review. In general, the threshold values are first set using heuristics found in the literature. They are refined by the quality team following an iterative process that consists of evaluating some systems and discussing the results with the project teams. The satisfaction-level rules could involve more than one metric. For example, rules for the code readability involve cyclomatic complexity, size, and comment metrics.

For the criteria related to the presence of anti-patterns, a detection mechanism is first applied. Two types of mechanisms are used, both are based on metrics: comparing entity-metric values to relative or absolute thresholds and comparing the metrics values between pairs of entities. An example of the first type is the detection of the presence of a *blob*. When the value of a complexity metric of a class exceeds the average value of the system multiplied by a coefficient, the class is a considered to be a *blob*. The second type of detection mechanism is used for example for the anti-pattern *copy-paste*. A similarity distance is calculated on metric-value vectors representing pairs of methods. If this distance is lower than a threshold, the anti-pattern is considered to be present. When the presence of an anti-pattern is established in an entity (method, class, or system), this entity receive a score of 4 for the presence of the anti-pattern. Otherwise the score is 1.

For the standard-conformance criteria, the score is attributed according to satisfaction rules that map the number of violation to scores following predefined thresholds. In the case of architecture conformance for example, the architecture of a system is reverse-engineered from its source code and compared against the architectures described in the design documents. The numbers of potential violations, as described in [18], is then derived. Figure 5 shows an example of architectural violation. The boxes are components (main packages/modules) and the arrows are reference links such as invocation relationships. The arrow highlighted by the ellipse indicates that the component Service is using Applets, while in the designed architecture Applets uses Service but the inverse link is forbidden.
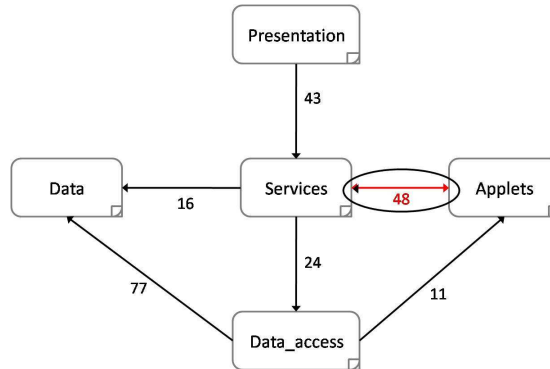
9

Figure 5: Example of Architectural Violation (highlighted in the ellipse).

In the particular case of unit-test coverage, a coverage percentage is derived from the tests cases according to the guidelines provided in [19]. The percentage is then converted to a score using again satisfaction-level rules.

Criteria are evaluated at different levels of granularity. For example, code readability is evaluated at the method level whereas the architecture conformance is evaluated at the system level. A criteria, measured at a certain level, could be used to derive a score at the upper level as a weighted average. For example, the code-readability scores of a class is calculated as the average of the readability scores of its methods weighted by their respective sizes in LOC.

Criteria scores are used to calculate factor scores as it is described in the next section. They are also used to point out problematic entities after the evaluation.

2.2.3. From Criteria to Factors

At this step all the criteria that are used to assess the quality factors have scores in the 1-to-4 scale of satisfaction. For each factor, the result of the application of GQM is a set of influencing criteria. Consequently, the factors are evaluated as a weighted average of the corresponding criteria scores. For example, maintainability is derived from the following criteria: presence of some anti-patterns (*copy-paste* and *recursive programming*), structural complexity, code-readability, conformance to some standards, and size. The highest weight

10

is assigned to code readability (5) whereas the lowest one is assigned to the one of the presence of the *recursive programming* anti-pattern.

Similar to the threshold values, weights are defined using heuristics and then refined based on experiments and feedback from the project teams. For example, the weight of code readability was set to 3 in the first version of MQL. Some criteria were even removed because their impact was not judged to be significant. This was the case of the *poltergeist* anti-pattern that was part of the calculation of the maintainability score and was removed in the third version of MQL.

2.3. Lessons Learned

The experience of implementing MQL confirmed most published observations. We discuss here some of the most interesting lessons that were not reported before or that slightly differ from common beliefs.

Technical staff and adoption. Previous studies reported that the project technical staff was reticent to the adoption of measurement programs ("Big brother" syndrome). The experience of MQL showed that this was not the case. Developers were receptive to the program and perceived it as a quality control tool rather than a management tool. This was probably because MQL was designed and introduced by technical staff.

Project managers and adoption. Project managers with little technical expertise were difficult to convince on the ROI of MQL. The link between improving software quality and reducing maintenance effort was perceived to be unclear.

Process change management. MQL was an intrusive program with respect to the software development process. A planned project focused on process change management was considered vital. Such a project made the initial adoption overhead a part of the company strategy. To this end, long-term support from the upper-management was a necessity.

Impact on the adoption of new practices. When implemented, MQL helped in the adoption of new technologies and practices. After MQL, model-driven engineering and code generation practices were easier to implement, though a substantial effort .

3. MQL Impact Study

The implementation of the MQL measurement program was done in two phases. In the first (pilot) phase, some project teams used MQL on a voluntary basis. In a second phase, the use of MQL was generalized to almost all the project teams. The study described in this paper is based on the data collected during the first phase.

We present the objectives of the study, its hypotheses, and its variables. We describe and justify the data collection and analysis. In addition to an objective evaluation, we conducted a subjective evaluation using a questionnaire to compare our study results with the perception of the project and quality teams.

3.1. Objectives

The objective of this study is to evaluate and report on the impact of the MQL measurement program on a set of indicators of software development process and product quality. We investigate three sets of quality indicators corresponding to the long-term objectives of MQL defined in Section 4:

- product quality factors, as defined by the DSIV;

- corrective maintenance effort;

- and, code cognitive complexity.

3.2. Hypotheses

For each of the three sets of quality indicators, we define a hypothesis to be tested with respect to the use of the MQL measurement program. For each

hypothesis, $H_{QP}i$, we provide its formulation and rationale, based on the quality analysis process in Figure 2:

- $H_{QP}1$: *MQL-monitored systems have higher quality factor scores.* The goal of implementing the MQL measurement program is to improve the different quality factors of the delivered systems through, for example, refactoring and the removal of code and design smells. Thus, we study whether the MQL measurement program actually helps to reach this goal;

- $H_{QP}2$: *MQL-monitored systems require less effort on corrective maintenance.* As stated in Section 1, before the MQL measurement program was designed and implemented, many defects were detected after release, thus requiring an important correction effort. We refer to this effort as corrective maintenance effort. We therefore study the variation of corrective effort distributions among systems using MQL and ad-hoc approaches;

- $H_{QP}3$: *MQL-monitored systems have less complex and more documented code.* One of the objectives of the MQL measurement program is to ease evolution due to frequent changes in requirements by reducing code complexity and increasing the number of comments in the code. This is particularly crucial for SNCF because many components are subcontracted and there is an important personnel turnover.

A null hypothesis $H_{QP}i_0$, stating that there is no difference between the two groups of systems (MQL versus ad-hoc approaches), is defined for each alternative hypotheses $H_{QP}i$. Following standard procedures [20], we study the rejection of these null hypotheses as a means of supporting the $H_{QP}i$ hypotheses.

3.3. Variables

We choose the following variables as means to assess whether the previous hypotheses are verified or not.

13

Independent and Mitigating Variables. The main independent variable in our study is the program used to monitor the quality of the systems, either MQL or ad-hoc, referred to as quality approach (QP). This variable takes value 1 for systems monitored with MQL and 0 for ad-hoc systems.

There are, however, mitigating variables that could affect the impact of QP on the indicators. These variables include system size (LOC), team size (T_S), team maturity (T_M), and team nature (T_N). In addition to using these variables for sampling the systems to be analyzed, we investigate their interaction effects with QP.

We chose a reference period (RP) to measure these variables on each studied system to collect comparable data. The period varies across selected systems and corresponds to the development cycle of last completed versions of the systems, from the requirement phase to the release. System size is measured as the number of lines of code of the delivered release of the system at the end of RP. The team size is the average number of team members during RP. The team maturity is the median expertise of the team members during the RP. The expertise of each member corresponds to the number of years of practice for the technology and language used. It is measured on a 0-to-4 scale where 0 ("beginner") denotes "less than one year" and 4 ("expert") denotes "more than four years". Finally, "team nature" specifies whether the system is developed by an internal team or a subcontractor.

Dependent variables. Dependent variables are interesting indicators that may be influenced by the independent variables. We select nine dependent variables corresponding to the three stated hypotheses, including six quality factors, one factor related to corrective-maintenance effort, and two code cognitive complexity factors.

Related to quality factors (first hypothesis), for consistency with the objectives of MQL, we select the six quality factors defined by DSIV: maintainability, evolvability, reusability, robustness, testability, and architecture quality (see Section 2.2.

14

Using the factors that the MQL measurement program tries to optimize could be seen as a bias. We further discuss this threat to the validity in Section 5. However, the fact that MQL purports to improve these indicators does not automatically imply that it succeeds in doing so. Indeed, the relationship between MQL corrective actions and the factors is not straightforward. Thus, one important objective of our study is to verify whether applying MQL will lead to significant improvements.

Related to corrective maintenance (second hypothesis), we use the corrective maintenance effort as a dependent variable. This effort is measured as the proportion of effort dedicated to fault correction during RP. Maintenance effort (time) is divided in three categories: design, development (new features, excluding correction), and correction. For each category, we define a variable:

- PDsT: percentage of time spent in design;

- PDvT: percentage of time spent in development, including refactoring;

- PCoT: percentage of time spent in fault correction;

with PDsT + PDvT + PCoT = 100% for a system.

Time distribution over the categories is obtained by analyzing time sheets of project teams, covering RP. As we are interested in corrective maintenance, only PCoT is compared across quality approaches.

Related to code cognitive complexity (third hypothesis), we use two dependent variables based on our assumption that code complexity and comments are essential factors for facilitating evolution:

- PCUC: proportion of complex and unstructured code. PCUC is computed as the percentage of methods/functions with an essential complexity ev [17] higher than 10;

- PCoC: proportion of commented methods/functions.

15

3.4. Data Collection

We use random sampling to create the treatment group: 22 systems selected randomly from those using MQL measurement program. To create the control group, we use matched-pair sampling [21]: 22 systems where ad-hoc approaches were used by the project teams to assess and control quality. These systems were selected to be comparable with MQL systems in terms of the number of lines of code, team sizes, team maturities, and team natures.

The population of systems using MQL from which we selected randomly our treatment group is composed of systems covering a representative sample of the types of systems that are developed under the supervision of DSIV. The adoption of MQL by project teams was done on a voluntary basis. Teams were encouraged to use MQL through tutorials given by the quality team. After discussions with these teams and DSIV quality staff, we ensured that there was no particular reason for systems using MQL to have had a higher quality prior to the use of MQL in comparison to non-MQL systems.

For the selection of the control group, it was difficult to find perfect matches for system pairs as required in random-paired sampling because of the limited number of systems at hand. However, we obtained comparable groups for the team size (TS), maturity(TM), and nature (TN) as shown in Figure 6, where similar distributions can be observed across QP groups. In this figure, the frequencies (or counts) of the systems are represented as pyramids where the left-hand (respectively, right-hand) sides give the distributions of the control (respectively, experimental) group systems over the four mitigating factors. For system size (LOC), a single large system that does not use MQL introduces an important difference in size between the groups. We confirm that this difference is not statistically significant by performing a mean difference t-test.

Tables 1 and 2 gives some descriptive statistics of the dependent variables. It shows that data is missing for the three effort variables, where we succeeded in collecting data for 27 systems only. For the remaining 17 systems, it was difficult to establish the effort distribution accurately. Most of them are external systems (11) for which subcontractors communicated incomplete data. For the missing
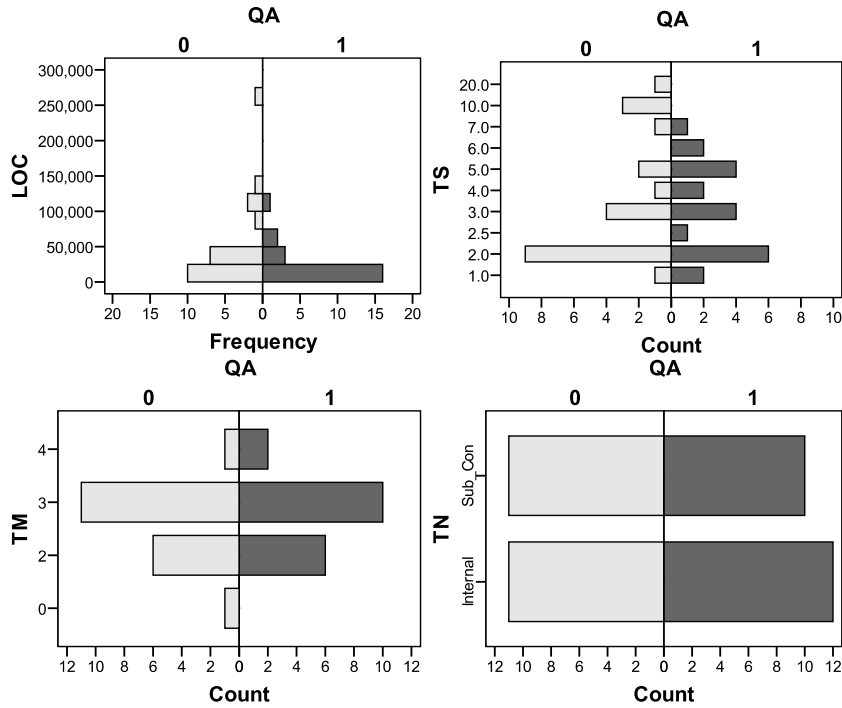
Figure 6: Influencing Elements Distribution among QP Groups

internal systems, the effort reported was ambiguously labeled.

We performed normality tests on the extracted data to select the more appropriate statistical test for each dependent variable. Robustness is the only variable that is not normally distributed.

3.5. Analysis Techniques

In case of normally distributed dependent variables (all but robustness), we use a Student t-test after checking equality of variances (Levene's Test). We tested robustness, non-normally distributed, with the Mann-Whitney Test. We chose a level of confidence of 95% to reject the null hypothesis, *i.e.*, a significance (Sig.) less than 0.05. We also studied the distribution of data in the samples to assess the practical significance of differences in central tendencies between systems using the MQL and ad-hoc approaches.

	# systems	Min	Max	Mean	Std Dev
Maintainability	22	2.51	3.89	3.42	0.37
Evolvability	22	1.87	3.89	3.34	0.53
Reusability	22	1.23	4.00	3.50	0.70
Robustness	22	2.44	4.00	3.87	0.33
Testability	22	1.72	3.82	3.07	0.58
Archi_qual	22	1.00	4.00	3.45	0.70
PCUC	22	0	16	5.27	5.43
PCoC	22	0	28	13.03	6.69
PDsT	15	0	0.60	0.24	0.17
PDvT	15	0	0.96	0.59	0.25
PCoT	15	0	1	0.17	0.25

Table 1: Descriptive Statistics of the Experimental Group

Our study involves multiple tests, one for each of the nine dependent variables. We correct the p-values using the Benjamini-Hochberg correction [22]. This correction is based on controlling the false discovery rate (FDR)—the expected proportion of false discoveries among the rejected hypotheses. In general, the correction provides a good balance between discovery of statistically significant differences and limitation of false positive occurrences. The calculation of the correction is as follows. First, p-values of the nine tests are sorted from the largest p-value to the smallest one ($p_9 \geq p_8 \geq ... \geq p_1$). The largest p-value p_9 is not corrected. Then, each remaining p-value p_i is corrected by considering its position i and the adjusted $(i + 1)^{th}$ p-value p'_{i+1}; formally:

$$p'_9 = p_9;$$
$$p'_i = \min(p'_{i+1}, \min(\tfrac{9}{i} \times p_i, 1)).$$

To study the interaction effects between the quality approach and the four mitigating variables (project size, and team size, maturity, and nature), we use a general linear model (GLM) test [23].

Finally, we compare our results of the impact of the MQL program with subjective analyses from the project teams that MQL. We use a questionnaire

	# systems	Min	Max	Mean	Std Dev
Maintainability	22	2.35	3.54	2.85	0.32
Evolvability	22	1.85	3.45	2.61	0.47
Reusability	22	1.23	4.00	2.63	0.91
Robustness	22	2.01	4.00	3.04	0.70
Testability	22	1.69	3.47	2.31	0.48
Archi_qual	22	1.00	4.00	2.45	0.89
PCUC	22	0	45	12.29	12.38
PCoC	22	0	27	13.57	6.59
PDsT	12	0	0.70	0.22	0.20
PDvT	12	0	0.90	0.45	0.25
PCoT	12	0.05	1	0.33	0.31

Table 2: Descriptive Statistics of of the Control Group.

to collect the answers of the project teams to the following questions, using a three-point Lickert scale ("Yes", "No", "I don't know"):

1. Does MQL impact the development effort?
2. Does MQL impact the maintenance effort?
3. Does MQL impact the number of defects?
4. Does MQL impact the subcontractor-management effort?
5. Does MQL impact the training effort?

For each question, project members could provide an additional explanation.

4. Results and Discussion

We now present the study results for the three hypotheses on the use of the MQL measurement program. When analyzing the corrective-maintenance effort, the sizes of the treatment and control groups change due to missing values.

4.1. Hypothesis $H_{QP}1$

Our first hypothesis is that MQL-monitored systems have higher quality factor scores than the other systems. As shown by the box-plots in Figure 7, for all quality factors, MQL systems show better scores than ad-hoc systems. Moreover, scores of the MQL systems show more predictable quality levels with narrower box-plots indicating less variability in quality among MQL systems compared to those using ad-hoc approaches.
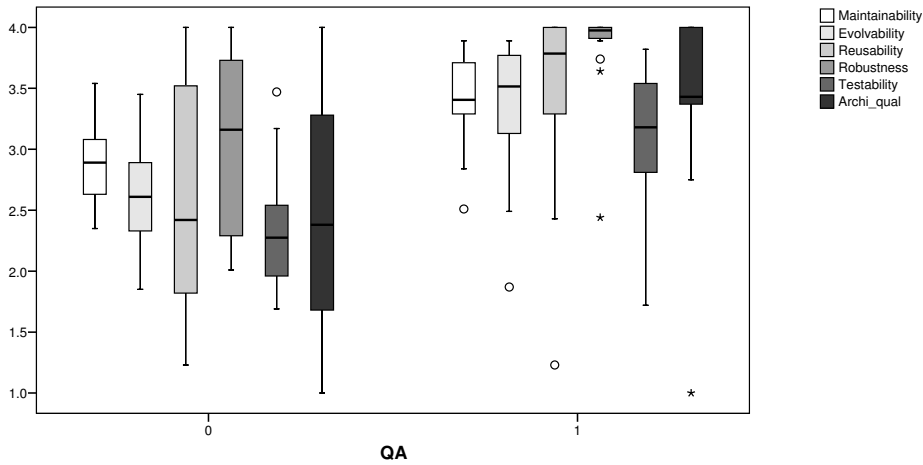

Figure 7: Score Distribution for Quality Factors for MQL and ad-hoc Systems.

Only one system has scores lower than the median of ad-hoc systems for each quality factor (it appears as an outlier in the figure). After discussion with the involved project and quality teams, we understand that this system was initially a very large and poorly structured system using an ad-hoc quality monitoring. The evaluated version is the result of a restructuring period using MQL. Although several improvements took place, they were not sufficient to significantly improve the overall quality of the system.

The impact of MQL is statistically significant with a confidence of nearly 100% (bold-face in Tables 3 and 4) for all quality factors. The corrected p-values allow us to reject the null hypothesis $H_{QP}1_0$ and the box-plots support the alternative hypothesis that MQL significantly improves the quality of the

	t-Test				
	t	df	Sig. (2-tailed)	Mean Difference	Standard Error Difference
Maintainability	-5.377	42.000	**0.000**	-.56364	0.105
Evolvability	-4.809	42.000	**0.000**	-.72636	0.151
Reusability	-3.584	39.489	**0.001**	-.87591	0.244
Testability	-4.743	42.000	**0.000**	-.76136	0.160
Architecture Quality	-4.136	39.626	**0.000**	-1.00000	0.242

Table 3: Significance testing for Mean Differences of Normally-Distributed Quality Factors.

	Mann-Whitney Test			
	Mann-Whitney U	Wilcoxon	Sig. (2-tailed)	Z
Robustness	71.000	302.000	**0.000**	-3.643

Table 4: Significance testing for Mean Differences of Robustness.

systems. This improvement is also practically significant because, for almost all of the factors, ad-hoc systems show scores that average between 2 and 3 (acceptable with modifications), whereas MQL system scores are consistently greater than 3 (acceptable).

4.2. Hypothesis $H_{QP}2$

For the second hypothesis, we want to check if MQL-monitored systems require less effort on corrective maintenance than their ad-hoc counterparts. MQL systems require indeed relatively less corrective maintenance than ad-hoc systems: on average 12% of the total development effort for MQL systems compared to 27% for ad-hoc systems, as shown in the pie charts in Figure 8. Though development and fault correction effort percentages change, design effort remains almost the same across QP groups. The box-plots in the same figure show that all but two MQL systems show a small proportion of corrective mainte-
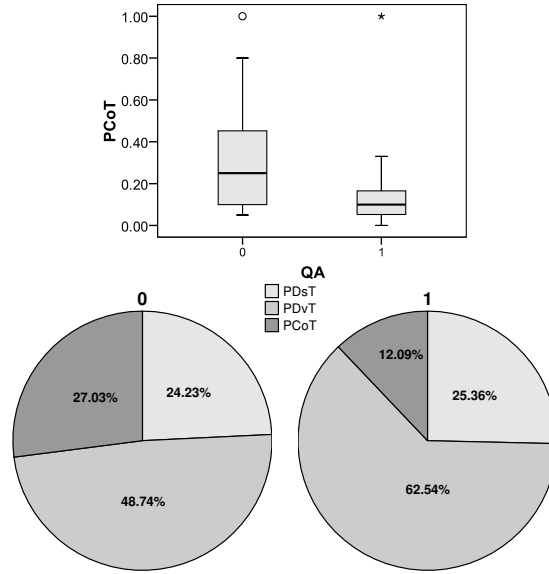
Figure 8: Effort Distribution For MQL and ad-hoc systems.

nance (less than 20%). Ad-hoc systems show higher variances in their results. The average difference between the two groups (15%) can be considered practically significant, especially for large systems with long and resource-consuming development cycles.

Although the effort distribution is largely in favor of MQL systems, the p-value is slightly higher than the significance threshold set for rejecting the null hypothesis (Sig. 0.054), as shown in Table 5. Considering the objectives of our study, which is about assessing the benefit of using a method in a company, and not establishing scientific truth in conservative terms, we can reasonably conclude that the proportion of the effort dedicated to corrective maintenance will more likely than not decrease when using the MQL measurement program.

	t-Test				
	t	df	Sig. (2-tailed)	Mean Difference	Standard Error Difference
PCoT	2.019	25	**0.054**	0.138	0.068

Table 5: Significance testing for Mean Differences of Correction Effort.

4.3. Hypothesis $H_{QP}3$

The goal of our third hypothesis is to investigate whether MQL-monitored systems have less complex and more documented code. We found that with the MQL measurement program, the percentage of complex and unstructured code drops significantly from, on average, 13.82% to 5.27%. This difference is visible in the box plots of Figure 9. More than a third of MQL systems contain less than 10% of unstructured code. The difference between the groups is practically significant. Indeed, an increase by more than 8% of unstructured code in a large system is likely to lead to significantly higher maintenance costs, especially if that unstructured source code is frequently modified. Regarding the proportion of commented code, we observe only a negligible difference (nearly the same median).
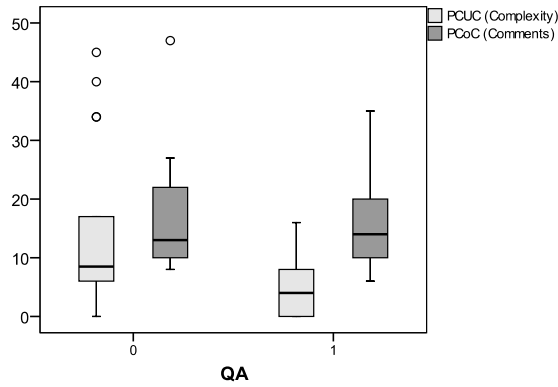

Figure 9: Percentages of Complex and Commented Code in MQL and ad-hoc Systems.

While the improvement in code complexity is statistically significant to reject the null hypothesis $H_{QP}3_0$ (even after correction), the improvement in terms of commented code is not, as shown in Table 6. We believe that MQL project teams are more careful about code complexity because they know that complexity metrics are used to compute the scores of some of the reported quality factors. This is not the case for the comments, which are perceived to be of minor concern to the project teams. Another explanation is that in the tools used in SNCF to extract metrics, the proportion of commented code for a program is

23

calculated on the basis of the presence of comments inside the method/function bodies. This does not consider other forms of comments such as class comments and our measurement may therefore be incomplete.

			t-Test		
	t	df	Sig. (2-tailed)	Mean Difference	Standard Error Difference
PCUC	2.863	28.227	**0.008**	8.545	2.985
PCoC	-0.142	42.000	**0.888**	-0.364	2.557

Table 6: Significance testing for Mean Differences of Percentages of Complex and Commented Code.
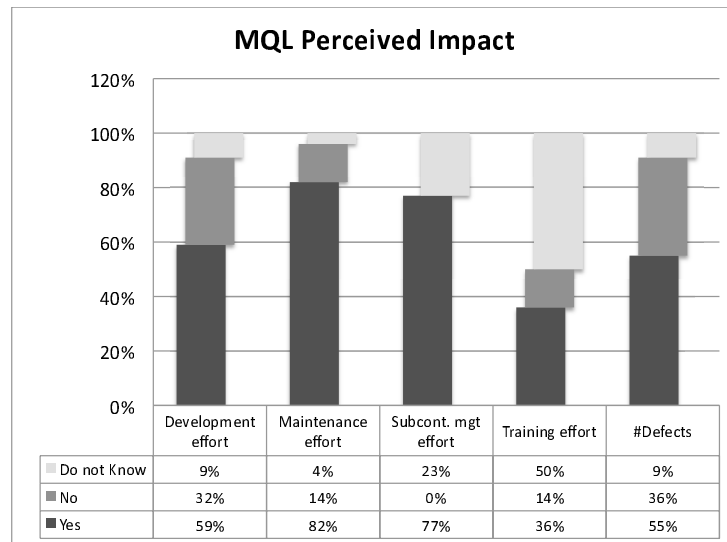
4.4. Subjective Evaluation

Figure 10: Subjective Evaluation Results

Figure 10 summarizes the results of our survey of the perceived impact of the MQL measurement program by project teams. 82% of the respondents stated that the MQL program *does* impact the maintenance effort. When reading their explanations, a large majority of teams stated that the maintenance effort

is lower when using MQL. With a similar proportion (77%), they reckoned that MQL makes it easier to manage subcontractor relationships.

For the other questions, more than half of the respondents agrees that MQL does impact the number of defects (55%) and the development effort (59%). In the first case, they indicate that the number of defects decreases when using MQL. In contrast with the diminution of maintenance effort, respondents note an increase in development effort. This could be explained by the overhead introduced by MQL.

Finally, unexpectedly, few respondents found that MQL facilitated the training of new employees. One could conjecture that MQL, proposing documented best practices, would help training new employees.

4.5. Conclusion

The study results confirm that systems monitored with the MQL measurement program obtain better quality scores for all the variables on the studied systems. This improvement is statistically significant for quality factors and code complexity. It is however slightly above our significance threshold ($0.054 > 0.05$) for corrective maintenance. We believe that the indicators that are significantly improved are those on which a new measurement program can have a rapid and measurable impact, over the chosen reference period, such as code quality. Conversely, a longer period of use of the MQL program may be needed before a larger impact can be observed on corrective maintenance. Indeed, defects that appear in a version may be due to design decisions made in earlier versions or incorrect previous changes.

To refine our understanding of the impact of MQL, we also studied the interaction effects of the four mitigating variables defined in Section 3.3. We have not found any significant interaction effect after removing outliers, thus suggesting that the impact of MQL does not depend on these variables. Our results were confirmed by a subjective evaluation.

5. Study Validity

Regarding construct validity, when mapping the hypotheses to variables, we selected quality factors defined by SNCF rather than defining new study-specific variables. This choice has two advantages: (1) measurement data is already available in the SNCF project repository and (2) the selected factors are those that MQL is intended to improve. However, although these indicators are documented and systematically measured, they could be significantly improved to better capture the aspects they measure.

When establishing the hypotheses, we were particularly careful to avoid any experimenter expectancy by conducting the study at the University of Montreal in collaboration with Simula Research Laboratory, while the co-author from DSIV participated only in data collection and results interpretation. SNCF project teams and external sub-contractors were not informed beforehand of the study. The data was collected as part of a SNCF-wide initiative.

We did identify a possible threat to internal validity, that was unfortunately not avoidable in our context. Indeed, 12 of the 22 MQL project teams have a permanent quality analyst attached to the teams. In contrast, only one team of the ad-hoc systems has such an analyst. It is, however, difficult to assess how serious this threat is.

We are confident about the validity of our statistical analyses. We followed a rigorous analysis protocol that enabled us to choose the most appropriate statistical technique for each hypothesis considering the data properties. The only threats that may affect our conclusions are "fishing and high type I error rate" and errors in data measurement. For the first threat, since we performed multiple tests (nine variables), we corrected the level of significance using the Benjamini-Hochberg correction procedure [22]. All the significant tests remained significant after the correction.

In the case of measurement, robustness is calculated on the basis of the unit tests actually performed. The data was provided by the project teams, and in a few systems, it was given in a format that made it difficult to determine

the actual coverage. This difficulty in analyzing the data explains in large part some surprising results obtained for robustness. Indeed, robustness was the only quality factor with non-normal distribution with almost all the MQL systems having a score close to 4.

Finally, we consider that the selected systems are representative of those developed by SNCF in terms of variability in sizes and domains and thus are not a threat to the external validity of the study. However, as some data was missing for the effort distribution, the treatment and control groups were less balanced in terms of the number of systems (15 *vs.* 12) and the distributions of team maturity and team nature, as shown in Figure 11. This is a possible explanation for the lack of statistical significance obtained for the corrective maintenance effort in Section 4.2.
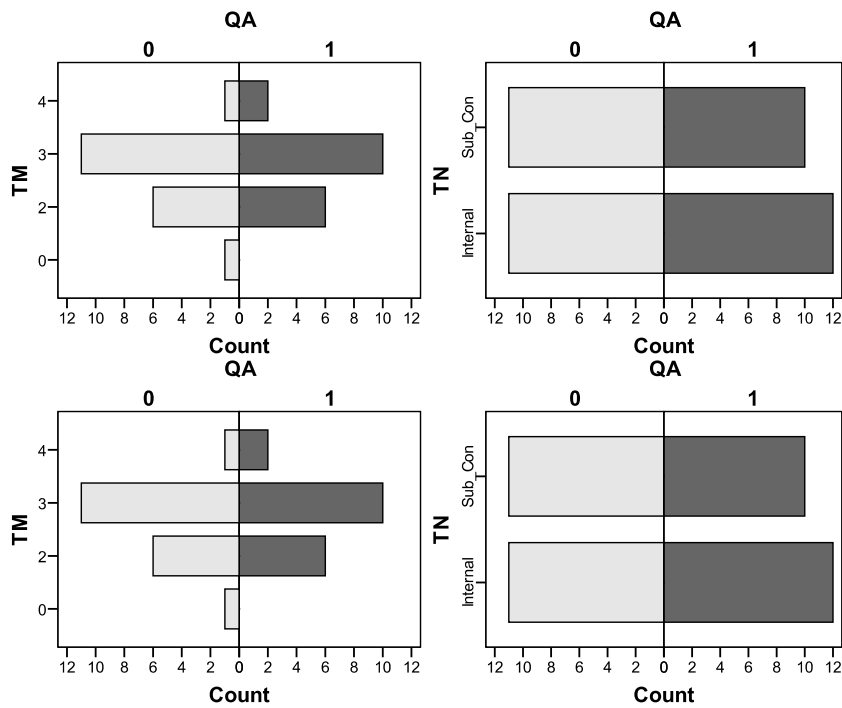

Figure 11: Influencing Elements Distribution among QP Groups for the 27 Projects with Effort Data

More generally, SNCF itself is representative of many large companies in the

transportation sector and we therefore expect that the obtained results can be interesting and generalizable to other organizations.

6. Related Work

Requirements, guidelines and principles for implementing measurement programs have been discussed in many publications (see for example [24]). These principles are usually based and illustrated through concrete examples of measurement programs, such as for the program of MOTOROLA described in [25]. There are, however, few contributions on the evaluation of measurement programs in industrial contexts. The existing contributions are in general case studies. Very few are controlled empirical studies.

In case studies [5, 6, 7, 10], the experience of implementing measurement programs is discussed from different perspectives and lessons learned are reported. Examples of these lessons are the necessity of considering the motivation and the perception of the developers, the reduction of the overhead introduced by the program, and the careful use and dissemination of the collected data.

Closer to our work is the study described in [8]. The authors conducted a survey by means of a questionnaire. Using the collected data, they investigated the relationship between some organizational and technical variables, and measures of the program success. The obtained results confirmed many general beliefs, *e.g.,* data collected in the programs facilitate the process of decision making, and the involvement of the upper management in the programs as well as the project managers improves the organisational performance.

Another similar study is described in [9]. The authors evaluated a measurement program implemented in Samsung in a preliminary phase by applying it on nine projects. In this case also, the results suggested that the program does improve the software quality in the long run.

From the methodological point of view, the work presented in [26] is particularly interesting. An approach is proposed for measuring the performance of measurement programs. The approach was tested successfully on a sample of

measurement programs.

The main differences between the study reported in this paper and the ones discussed above is that we work on a much larger sample of projects (44), that also include a control group where quality is not monitored or controlled by a carefully planned measurement program. This enabled us to perform a quantitative and rigorous evaluation of the implemented measurement program.

7. Conclusion

We presented an empirical study following a rigorous methodology to assess the impact of a measurement program (MQL) on a set of product and project quality indicators. The study was conducted in an industrial context, on 44 real-life systems of the French-railway company SNCF. We specifically studied the impact of the MQL measurement program ("Mise en Qualité du Logiciel", French for "Quality Software Development") developed and implemented by the SNCF IT division DSIV, which consists of a continuous process that, for each system release, allows for semi-automated quality evaluations, comparisons with historical data, and suggestions of corrective actions.

The results showed that the use of MQL has an impact on all studied indicators. This impact is statistically significant for six quality factors (maintainability, evolvability, reusability, robustness, testability, and architecture quality) as well as for code complexity. The differences between the experimental and control groups are large enough to be of practical significance. Moreover, all the significant benefits observed for the studied quality aspects cumulate to bring an overall important improvement in the quality of delivered systems. More specifically, our study demonstrated that the MQL measurement program already meets the long-term objectives defined by SNCF. Indeed, quality factor scores are significantly better for projects using MQL. The proportion of effort dedicated to corrective maintenance was reduced when using MQL. Finally, although MQL does not improve the practice of commenting the code, our study showed that it helped to better master complexity of the produced code. More

generally, our study concretely supports the idea that measurement programs can have a significant, positive impact on the quality of software systems if put into place carefully and thoughtfully.

As expected for any industrial study, we faced many challenges, mainly related to data collection. In particular, when dealing with sub-contractors, it was difficult to validate the accuracy of the provided data. However, as explained in Section 5, most data was systematically collected from an existing central repository. We also made sure to prevent possible biases that may limit the generalization of the results, at least in the context of SNCF.

In addition to experimental results, this study highlighted possible improvements in the studied measurement program. A project has been set up to implement these improvements. First, more sophisticated quality models are defined using large sets of historical data available at DSIV. These models will capture the context of SNCF better than existing general models (aggregation of weighted metrics following ISO 9126 [12]). Second, the algorithms for code and design smell detection are refined and augmented. Finally, visualization techniques are developed to better display the analysis results and compare them with historical data.

Acknowledgments

References

[1] T. DeMarco, Controlling Software Projects: Management, Measurement, and Estimation, Yourdon Press, 1982.

[2] N. E. Fenton, S. L. Pfleeger, Software Metrics : A Rigorous and Practical Approach, Course Technology, 1998.

[3] T. Nakamura, V. R. Basili, Metrics of software architecture changes based on structural distance, in: Proceedings of the 11th International Software Metrics Symposium, 2005, pp. 78–87.

[4] M. B. Chrissis, M. Konrad, S. Shrum, CMMI : Guidelines for Process Integration and Product Improvement, 1st Edition, Addison-Wesley, 2003.

[5] S. L. Pfleeger, Lessons learned in building a corporate metrics program, IEEE Softw. 10 (3) (1993) 67–74.

[6] T. Hall, N. Fenton, Implementing effective software metrics programs, IEEE Softw. 14 (2) (1997) 55–65.

[7] J. Iversen, L. Mathiassen, Lessons from implementing a software metrics program, in: HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 7, IEEE Computer Society, Washington, DC, USA, 2000, p. 7040.

[8] A. Gopal, M. S. Krishnan, T. Mukhopadhyay, D. R. Goldenson, Measurement programs in software development: Determinants of success, IEEE Trans. Softw. Eng. 28 (9) (2002) 863–875.

[9] H. Lee, Y. Jang, An experience of implementing software metrics in an industrial environment, in: Proceedings of the Second International Conference on Software Engineering Advances, 2007, pp. 42–47.

[10] I. D. Coman, A. Sillitti, G. Succi, A case-study on using an automated in-process software engineering measurement and analysis system in an industrial environment, in: Proceedings of the 31st International Conference on Software Engineering, 2009, pp. 89–99.

[11] O. Beaurepaire, B. Lecardeux, C. Havart, Exploring industrial data repositories: Where software development approaches meet, in: Proceedings of

the 8th workshop on Quantitative Approaches in Object-Oriented Software Engineering, 2004, pp. 47–59.

[12] ISO/IEC, Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use, iSO/IEC 9126:1991(E). (December 1991).

[13] R. Basili, D. M. Weiss, A methodology for collecting valid software engineering data, IEEE Transactions on Software Engineering 10 (6) (1984) 728–738.

[14] S. Henry, D. Kafura, Software structure metrics based on information flow, Transactions on Software Engineering 7 (5) (1981) 510–518.

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[16] S. Hommel, Java code conventions, Tech. rep., Sun Microsystems (2000).

[17] A. H. Watson, T. J. McCabe, Structured testing: A testing methodology using the cyclomatic complexity metric, Tech. Rep. NIST Special Publication 500-235, Computer Systems Laboratory, National Institute of Standards and Technology (1996).

[18] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of the 20th Conference on Object Oriented Programming, Systems, Languages, and Applications, 2005, pp. 167–176.

[19] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, Computing Surveys 29 (4) (1997) 366–427.

[20] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, Experimentation in Software Engineering: An Introduction, 1st Edition, Kluwer Academic Publishers, 1999.

[21] D. S. Moore, G. P. McCabe, Introduction to the Practice of Statistics, 5^{tht} Edition, W. H. Freeman & Co., New York, NY, USA, 2006.

[22] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach to multiple testing, Journal of the Royal Statistical Society Series B 57 (1) (1995) 289–300.

[23] N. H. Timm, T. A. Mieczkowski, Univariate & Multivariate General Linear Models: Theory and Applications Using SAS Software, SAS Publishing, 1997.

[24] R. J. Offen, R. Jeffery, Establishing software measurement programs, IEEE Softw. 14 (2) (1997) 45–53.

[25] M. K. Daskalantonakis, A practical view of software measurement and implementation experiences within motorola, IEEE Trans. Softw. Eng. 18 (11) (1992) 998–1010.

[26] M. Berry, R. Jeffery, An instrument for assessing software measurement programs, Empirical Softw. Eng. 5 (3) (2000) 183–200.