Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality

Mathieu Lavallée, Pierre N. Robillard
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montréal, Canada
[mathieu.lavallee, pierre.robillard]@polymtl.ca

***Abstract*—How can organizational factors such as structure and culture have an impact on the working conditions of developers? This study is based on ten months of observation of an in-house software development project within a large telecommunications company. The observation was conducted during mandatory weekly status meetings, where technical and managerial issues were raised and discussed. Preliminary results show that many decisions made under the pressure of certain organizational factors negatively affected software quality. This paper describes cases depicting the complexity of organizational factors and reports on ten issues that have had a negative impact on quality, followed by suggested avenues for corrective action.**

***Index Terms*—Organizational factors, software quality, observational case study.**

I. Introduction and Related Work

Current studies on software development environments have focused on improving conditions for the software development team: implementation of appropriate processes, hiring skills, use of appropriate communication tools and equipment, etc. Among these software development conditions are factors from the organization. However, while such factors are perceived as important, there is little empirical evidence of their effect on the quality of software products [1], [2]. It can therefore be useful to present empirical data on the effects of organizational factors on the development team. What if a project's success is dictated more by the constraints imposed by the organization than by the expertise and methodology provided by the team?

Organizational factors include a wide range of contextual factors with a potential impact on the success or failure of software development projects. Early empirical work done in 1998 by Jaktman [3] showed that organizational values can have an impact on software architecture. She presents two interesting cases: the first revolves around a company who made software development dependent on the marketing department. As Jaktman writes: *"This resulted in software quality activities being given a low priority"* [3]. The outcome was a poor architecture, with a lot of code duplication and high error rates. This jeopardized the company's future, because *"maintenance problems prevented new products from being released quickly"* [3].

The second case presents a company who was officially committed to quality, but was ambivalent about how to implement it. Management promoted various quality approaches, but ultimately resources failed to materialize. The *"frequent changes to product priorities affected the code: incomplete implementation of software changes left dead code and code fragments"* [3].

These two cases show the possible relationship between high-level organizational values and low-level product quality. Unfortunately, studies reporting on the impact of organizational factors on software development are scarce. As Mathew wrote in 2007, *"it is surprising to note that academic research has largely ignored investigation into the impact of organisational culture on productivity and quality"* [1].

This paper answers the need for more empirical data through the presentation of observed cases describing how some organizational factors can impact software quality.

Section II describes the context and the approach used to collect data for the study. Section III presents the results of the study, while Section IV discusses the implications of the findings. Finally, Section V presents some concluding remarks and observations on organizational factors.

II. Methodology

A. Context

The study was performed on a large telecommunications company with over forty years in the industry. Throughout the years, the company has developed a large amount of software, which must be constantly updated. This study follows one such project update.

The outcomes of this study are based on observation of a software development team involved in a two-year project for an internal client. The project involved a complete redesign of an existing software package, which we will call the Module, used in the company's internal business processes.

The technical challenge of the Module is its distributed nature: it includes legacy software written in COBOL, Web interfaces, interactions with mobile devices and multiple databases. Its purpose is to manage work orders. To do this, it needs to extract data from multiple sources within the

enterprise (employee list, equipment list, etc.) and send it to multiple databases (payroll, quality control, etc.).

The project was a second attempt to overhaul this complex Module. A first attempt was made between 2010 and 2012 but was abandoned after the fully integrated software did not work. Because this project was a second attempt, many specifications and design documents could be reused. Accordingly, the development has essentially followed a waterfall process, as few problems were expected the second time around. This second attempt began in 2013 and finished in December 2014.

B. Data Collection

The study is based on non-participant observation of the software development team's weekly status meetings. These meetings consisted of a mandatory all-hands discussion for the eight developers assigned mostly full-time to the project, along with the project manager. They also involved, as needed, developers from related external modules, testers, database administrators, security experts, a quality control specialist, etc. The meetings involved up to 15 participants, and up to five stakeholders through conference calls.

The team discussed the progress made during the previous week, the work planned for the coming week and obstacles to progress. The problems raised concerned resources and technical issues. Few decisions were taken at these meetings, the purpose being to share the content of the previous week's discussions between developers, testers, managers, etc.

A round-table format was used, where each participant was asked to report on their activities. The discussions were open and everyone was encouraged to contribute. When a particular issue required too much time, participants were asked to set another meeting to discuss it. Meetings lasted about an hour.

The data presented in this study were collected over ten months during the last phase of the two-year project. It is based on 36 meetings held between January and November 2014. The same observer attended all the meetings and took note of who was involved in each interaction, the topic being discussed, and the outcome. An interaction is defined as a proposition or argument presented by one or more team members. A typical interaction takes between 5 seconds (e.g. "yes, I agree with you") and 30 seconds (e.g. the presentation of a solution by many team members). A topic (e.g. "should we deploy on Thursday or on Saturday?") could be discussed over multiple interactions (e.g. "yes because…" and "no because…"), and sometimes ended with an outcome (e.g. "we will deploy on Saturday").

C. Validation

The weekly status meetings provided a lot of information about the developers and the product. However, they provided only the perspective of the meeting participants. To better understand how the organization operates, the observer interviewed two managers from different departments (operations and marketing) on July 22nd in order to obtain their views. This validation was performed in a semi-structured interview. The observer asked the two managers if they agreed or disagreed with the preliminary conclusions made so far.

A second validation was performed at the end of the study. Since it was a non-participant study, the subjects observed were not aware of the conclusions reached by the researchers. A presentation was therefore made in December 2014 in order to confirm or refute the conclusions of the researchers. Results of the validation, when conflicting with our interpretations, are presented in the following and at the end of each case discussion.

Another validation was performed with an unrelated public sector software development department in December 2014. The thirteen workers were presented with the conclusions from this study and asked to rate whether the conclusion applied to their situation. For each of the cases identified in this study, the participants confirmed the occurrence of the issues within their organization.

D. Threats to Validity

The main issue was that the non-participant observer was not familiar with the technical terms used by developers, a common issue for non-participant observation studies [4]. The observer was present in the organization only during the weekly meetings, which means he was not aware of the interactions taking place outside the meetings. The observer was therefore not always up to date on what was going on. However, several problems recurred many times during the seven-month study, and were thus easy to follow and understand. The issues presented here are glaring problems that were not subject to interpretation and were all confirmed by the two managers during the validation interview.

As a single case study, generalization of the issues identified can be disputed. However, given the current small number of non-participant observational studies in software engineering, it is surmised that the data collected in this study remain interesting for future researchers. Additionally, given that the issues presented were confirmed during the validation process and in the literature review, it is believed that these issues are relevant for multiple contexts.

III. OBSERVATONS

The Core Team consisted of one manager and eight developers. However, it is surprising how many people are needed to interact with the core team to understand, develop, test and deploy the Module successfully. As shown in Figure 1, the core team interacted with no fewer than 45 people in at least 13 different teams during the ten month study.

Figure 1 also shows the relationship between the Core Team (shown in the middle) and the other teams involved in the project. Each team is represented by an oval shape with a descriptive label and contains the numbers and titles of the interacting members. For example, the Quality Assurance team, third from the top, clockwise, involved a quality tester and a quality manager interacting with the Core Team.

The innermost oval shape represents the Core Team, whose members were present at most meetings. The frequency of interactions with the Core Team is described by the dashed concentric circles. The second concentric

circle shows the people who came to most (i.e. at least half) of the meetings. The third concentric circle represents the people who came to only one or two meetings, or who were present through a conference call. The outer concentric circle represents people who did not attend the weekly meeting during the study, but who were referred to during discussions and contacted outside the meetings. For example, the quality tester from the Quality Assurance team attended seven meetings, and the Core Team members referred a few times to the quality manager's requests during meetings.

As the study focused on late phases of the development project, the teams in the second circle are mostly related to testing activities. In the early stages of the project, the operations department, being the internal client of the project, was much more involved.

Table I presents the organizational issues and their impacts on quality as observed during the study and confirmed during the validation. The following subsections detail each issue. The generic cases for each issue are discussed in Section IV.

A. The "documenting complexity" issue

This company has been in the telecommunications business for many decades. The objective of their software development department is to support the main business process and therefore the main concern is to provide software applications. Much of the existing code is poorly structured and difficult to understand.

Over the years, the company's legacy code has become very large, very complex and, like a lot of older software, is poorly documented.

Over time, the organization has encouraged software engineering practices to ensure that new software has sufficient documentation for maintenance and user support teams. However, most software modules are still poorly documented. To make matters worse, many developers have since left the company.

For the Module developed during this study, they have extensive documentation – thousands of pages – from the previous attempt to develop the software. This documentation is incomplete, and its quality is questionable as the previous implementation failed. Nobody knows exactly how it all works because of the size of the documentation. Despite their attempts to understand how the Module works, they assumed that some features are going to fall through the cracks.

Therefore, during the final stage of development, a senior developer was assigned to what they called the "crack", or the list of features that were initially overlooked. Some of

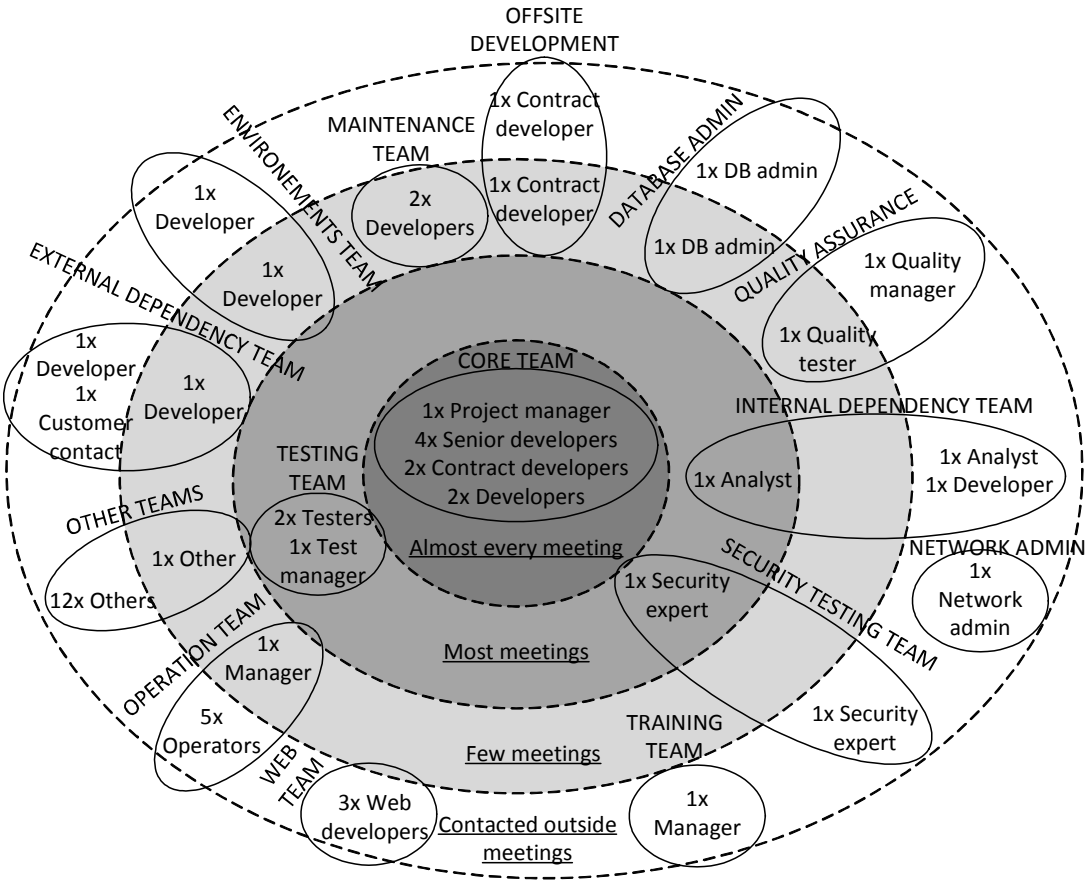

Figure 1. People and teams involved in the project, and how often they were in contact with the core development team.

these missing features were identified as "must-have" for operators and could very well be among the reasons for the failure of the previous project. For example:

- A number of daily maintenance scripts whose purpose is to maintain the database in a healthy state.
- A rarely-used user interface which aims to support the main business process when the main server is inaccessible due to connection issues.
- An archive feature used for quality monitoring.

These features escaped scrutiny, either because they were invisible to the operators of the Module, or because they were rarely used. The difficulty of documenting the complexity of the Module was that these functions were not noticed until late in the development.

The impact on quality was that some of these features were not fully implemented. For example, the rarely-used user interface will not have all the necessary features, but will be limited to a barebones interface. Other features that had fallen into the "crack" and were not identified as "must-have" were simply ignored. Developers have argued that they will be implemented in a future update.

B. The "internal dependencies" issue

The Module has many dependencies with other modules within the company's extensive legacy code. In addition, many other modules are simultaneously undergoing perfective maintenance. These interdependencies cause many issues since changes in the Module may affect other modules, and *vice versa*.

Priority decisions on module changes depend largely on the deployment schedules. For a module deployed later, the schedule must take into account the changes implemented on the previous modules. However, technical or business constraints may force the rapid deployment of certain features. For example, changes in the database must often be propagated early to ensure that the data structure will be

TABLE I. ORGANIZATIONAL ISSUES WITH AN IMPACT ON QUALITY AS OBSERVED DURING THE STUDY.

Observed issue	**Observed evidence**	**Observed quality flaw**	**Suggested corrective action**
Documenting complexity	Large amount of old poorly documented software. Difficulty in managing large amounts of documentation.	Some requirements are discovered late in the development process. Some undocumented features were not implemented; others were patched together very quickly.	Once a project is completed, the team must ensure that the "What" and "Why" of each software item are properly documented.
Internal dependencies	Lots of interdependencies between software modules. Conflicts between projects on the scheduling of deployment.	Compatibility between modules is patched quickly and haphazardly.	In the cases of parallel development of inter-dependent software modules set up a negotiation table to solve conflict between the development teams.
External dependencies	Long-term dependencies to third-party libraries. Change requests to third parties results in costly delays.	Contractual obligations with third parties force poor quality design choices in order to minimize change requests.	Make sure that the development team is aware of the CMMI-ACQ or ISO12207 processes for negotiating with third parties.
Cloud storage	Contractual obligations with third parties do not support vulnerability testing.	Vulnerability testing is performed late and quickly.	Make sure that testers are involved when negotiating with a third party for a potentially vulnerable software component.
Organically grown processes	Processes emerge as the need arise, usually after crises. Defects found by users are documented but do not reach developers. Environment setup process unclear for developers. Patches in testing follow a stringent process, even when nothing works.	Frontiers between processes hinder information exchanges; developers must work with missing details.	Plan organization-wide process reviews to detect isolated processes and to promote information flows between processes.
Budget protection	Cheaper to build a wrapper instead of solving the issue once and for all. A software item has twelve such wrappers.	Developers patch instead of fixing issues because fixing would cost too much.	Planned special budget items to support long lasting corrections or corrections that are likely to benefit many modules.
Scope protection	Explicit calls by team members to "protect the scope" of the project. Requirements are transferred to other projects as much as possible.	Project constraints means that teams must protect the scope of their project against change requests from other teams.	Projects with strict deadlines are risky, and should be carefully monitored to avoid last minute unplanned activities.
Organizational politics	Change request refused because the team did not contact the right person. Environmental issues resolved when the right manager was contacted. Inter-module issues resolved when upper manager applied pressure.	Managers and developers obtain better results by calling in favors from outside the software engineering process. Novices who do not know who to contact to obtain information will produce worse software.	Team members should maintain a careful balance between the flows of information within formal development processes and informal human interactions.
Human resource planning	The two developers assigned since the beginning of the project are contractual developers whose contract is about to expire.	The team will lose all knowledge of the beginning of the project, including details on the requirements and analysis phases. Documentation of the Module will be incomplete and/or inaccurate.	Team members should make sure that knowledge is appropriately distributed amongst them. For example, pair programming is a practice which can promote knowledge sharing.
Undue pressure	Managers and senior developers give direct orders and threats to the team.	The issuers of the orders and threats might not have all relevant information which could results in ill-defined priorities. Bypassing the normal team structure undermines its decision-making process.	Any intrusion into the team dynamics by outsiders should be done very carefully.

coherent in all the linked modules.

Change requests from these internal dependencies have an impact on the quality of the Module. These change requests are often implemented quickly and haphazardly, as they are rarely planned or budgeted. Even in cases where an analysis is performed, allowing the allocation of additional resources, deployment schedules are rarely modified, resulting in greater deadline pressures.

C. The "external dependencies" issue

The Module has many dependencies with third party modules, which are based on contract agreements. The Core Team must deal with these modules on a case-by-case basis, since each third party module has different contracts, practices and processes. Change requests ("CRs") are defined differently amongst the various third party contracts. For example, a minor change request on an external module might be considered billable for one third party supplier, or as covered by the maintenance contract for another. To make things worse, the quality of the work performed by third party suppliers fluctuates wildly. In many cases, the CR work had to be sent back to the supplier for rework, because it was not coherent with specifications, or of poor quality. Communication problems with third party suppliers were confirmed by many of the managers who had to deal with them.

These external dependency constraints force the team to make decisions that may have a significant impact on quality, in order to reduce costs and lead times. One practical example is the transformation of a variable type at the boundary between the Module and an external third party module from an enumeration to a free text type. From a quality standpoint, this is likely to introduce problems since the free text type cannot be validated, while the enumeration is self-documenting by the bounded and defined range of values.

So why was this decision made? Because each time an enumeration value needed to be modified, the third party supplier asked for a billable CR, which meant that changes to the enumeration type soon became prohibitively lengthy and expensive. In the short term, it was deemed cheaper to use a free text-type variable, even if it meant causing future hard-to-diagnose software defects.

D. The "cloud storage" issue

While not strictly pertaining to cloud computing, this issue is related to code hosted by other entities than the owner of the Module. In this observed case, some contractual agreements with these third party entities did not include all the planned test activities. This meant that some tests could not be performed until the contracts were renegotiated with the relevant clauses.

For example, the initial agreement did not allow vulnerability testing. The vulnerability tests were aimed at finding technical issues, like SQL injections, as well as overall robustness evaluation, like resistance to Denial of Service (DoS) attacks. The risk to quality is potentially huge. Without these tests, it is possible that a vulnerable code segment could remain in the Module, exposing the company to liability should a customer's account be compromised.

Vulnerability testing was thus performed very late in the development process, as the contracts needed reviewing by both parties. A quick superficial test found over 350 vulnerabilities in one third-party library which was already used in production, prompting one security expert to say that *"this was certainly written by junior developers"*.

E. The "organically grown processes" issue

The best practices of software engineering appeared in parallel with the company's growth. After all, the company has been around for decades, or about as long as software engineering itself. Software engineering processes were thus introduced organically, as the need arose or became apparent in a crisis. This resulted in "islands of formality": some software development teams within the organization follow a very clear and well-defined process, while others still work in a mostly ad hoc fashion.

For example, the quality control team follows a very well-defined process. The development team is more informal, but has a set of standard practices. And yet both processes evolved organically, so that that the quality control process is very different from the development process. This requires some coordination effort at the organizational level. The development team is aware that it must provide some documentation to the quality control team, but the developers do not know the level of documentation required.

These frontiers between processes resulted in some serious issues. For example, during the acceptance phase, the testers and the developers found that a requirement had not been met. This was a serious setback, as it meant that the developers and testers had to produce code and validate it in a rush. That missing requirement was to fix an issue with the previous version of the Module. The quality control team was aware of this requirement but failed or neglected to propagate the information to the development team.

Another serious issue was with the setup of testing environments. It took weeks for the testing environment to be set up, and the environment was never fully coherent with the specifications given because of misunderstandings between the developers' needs and the environment team's comprehension. It took many more weeks of back and forth iterations before the new environment could be used. Unfortunately, by the time the problems were solved, most of the development team's allowed time for this environment had elapsed and the testing environment was passed to another team. As one developer exclaimed during one of the meetings: *"We fought like dogs to get this environment and we can't even use it!"*

At the other end of the spectrum, the organization had introduced a new, very stringent process for the submission of code patches to non-development environments. This new process was introduced after insufficiently tested code ended up causing havoc in production. Though warranted, it causes a lot of complications when applied to patches in test environments. Code patches cannot be freely submitted to test environments because testers must be constantly aware of how the code changes within their testing environment.

The issue is that this procedure is applied even during environmental setup; that is, when code must be adjusted because the environment is being built for the first time. The new process did not take into account the development of new environments, which caused undue and frustrating delays.

F. The "budget protection" issue

The organization's culture puts an emphasis on staying on target, meaning deliveries on time and under budget. The pressure on the team is real: an observed example is the practice of patching instead of fixing. One specific external software item was in dire need of a fix. Unfortunately, a fix requires a formal CR—a costly and billable project estimated at about 20 person-days. The software development team instead chose to internally develop a wrapper in order to mash the data to address their need, a cheap patch which could be done for about five person-days.

Currently, that specific software item has about 12 such wrappers. This means that at least 12 other development teams chose the cheap patch solution instead of the costly fix, even if the one costly fix would have obviated the need for 12 cheap patches. In the words of the developers themselves, *"Eventually, one team will need to bear the cost of fixing this."* No one wants to be *that* team that will go over-budget, and be criticized for it.

The validation provided a mitigated point of view of this issue. The operations manager said budget was rarely an issue, probably because his team's work was directly linked to the company's main business process. The marketing manager, however, said budgets were tightly monitored and it was very difficult to argue for more time and money. Discussion with the Module's developers outlined that budget pressure was not the main issue, deadline pressure was. The team admitted that they had budget restrictions, but was much more concerned with limiting the scope in order to prevent deadline slippages.

G. The "scope protection" issue

A large part of the managers' duty is to protect the scope of the project, i.e. prevent modifications to the original specifications of the project. This is closely related to the internal dependency issue, as the scope of the project is mostly assailed by other projects that want to assign some of their related code changes to the Module. This may cause conflicts between development teams, since each team wants to protect its own project. The following scenarios are often put forward:

- **Requiring a change**: We ask you to adapt your code to our changes otherwise our project may fail integration testing;
- **Refuting a change**: We may ignore your code changes, since they are out of our project scope. If you really want it, we will bill your project for the changes.

The following example occurred one week before going into acceptance testing. The Module operators asked for a new feature, which would require some major changes. The team proceeded to:

- Convince the operators that it was not in the initial specifications, and that it may be implemented in a future project.
- Convince upper management that the change was not as minor as it first seemed, and could not be tested properly before acceptance testing, which would introduce some significant risks into the deployment.

Scope freezes exist within the organization, but are hard to enforce. The organization works in a very competitive domain: If a competitor provides a new service, the organization must support a similar service as soon as possible. Therefore, development priorities can be shifted overnight.

Additionally, compatibility between modules is often raised at the last minute and is therefore patched haphazardly, which may have a dubious impact on the resulting software product. This results in software defects at the boundaries between modules, where each side blames the other for the errors, lengthening the debugging process and making deadline slippage more likely. Therefore, the organization's culture of independent teams working within tight deadlines causes them to work mostly in competition with each other.

H. The "organizational politics" issue

Within an organization, the manager's most important tool is often his/her list of contacts.

For example, sometimes third party entities would refuse a CR on the basis that the change would violate the core functionality of their libraries. Inexperienced developers or managers would often accept these refusals at face value. However, employees with experience on the capability of the external libraries would often challenge these decisions. One manager confirmed cases where decisions challenged by experienced developers were conceded by the third party. In these cases, talking to the right person at the third party entity was the key factor of success.

Another example is illustrated by the problems that plagued the team for over two months concerning test environments. After nearly two months of stalling and non-functional test environments, a senior operations manager asked the Module manager to contact him each time a problem occurred with the environments. The problems were resolved within a week.

A similar problem, concerning inter-module communications, was resolved very quickly when the right person—the senior operations manager—applied pressure on the right people in charge of the other module.

Quality-wise, this means that many issues dragged on for weeks before being resolved, and that developers did not have as much time as they needed to correct the issues, which is likely to result in more patched-on code.

It also means that even a well-defined software engineering process would not resolve some of these issues, which are dependent on goodwill and personal contacts.

This kind of behavior is often deeply rooted within the organizational culture of the company.

I. *The "human resource planning" issue*

The composition of the development team evolved throughout the project. Some developers joined when their specific expertise was required, and left when it was no longer needed. This is typical of most development projects. However, what is peculiar in this case is that the only developers who have participated in the project since its beginning, the Module's "team historians" as defined by La Toza et al. ([5], cited by [6]), are two contractual employees whose contract is about to end. This means that the development team will lose all the knowledge of the first few months of the project just before deploying the application on the production server.

The team realized that this could be risky, and the contractual developers were kept until the deployment and asked to frantically write documentation in order to support maintenance activities after their departure. One contractual developer has been assigned a new experienced developer in order to transfer his knowledge of the project to him.

However, these knowledge transfer tasks were assigned on top of the usual activities of the contractual developers. During two weekly status meetings, the new developer simply states that he is *"waiting on the developer"* to give him the relevant information, hinting that knowledge transfer is not at the top of their priorities.

Human resources should be better planned for long-term software development. To quote Laurie and Kessler on the issue:

"It is not advantageous to have all knowledge in any area of a system known to only one person. […] This is very risky because the loss of one of a few individuals can be debilitating." [7]

At least the issue was handled before it was too late, but it could have been avoided altogether.

J. *The "undue pressure" issue*

As presented earlier, software development is not the main business process of the company. Its bread and butter are the operation of telecommunications services, and as such the most important department is operations. Consequently, operations have a lot of pull when it comes to coercing teams in other departments to perform specific tasks.

The observer has seen two occasions where operations personnel came to the weekly status meeting to put explicit pressure on the software developers. In one of the cases, the operations manager presented a clear threat: *"If you do not allow 100% of your time on* [a specific activity]*, I will be very disappointed. Very disappointed. Did I make myself clear?"* The operations department was clearly unhappy about the advancement of an activity and made its disappointment evident.

The "undue pressure" approach supposes that developers are not doing their best. Yet they were already working full time on their assigned tasks and were frantically putting the finishing touches on features that were about to be sent to testing.

The issue is that the pressure, which might be justified in some cases, must be applied in the appropriate context. Undue pressure from upper management or from other departments can coerce the team to work on subjects that are not optimal for the quality of the product, especially when management is not completely up-to-date with the project status.

IV. DISCUSSION

One of the main issues of observational study research is whether the observations can be generalized to other settings. In the case at hand, the following question can therefore be asked: Is this a one-of-a-kind organization, or does it represent a widespread situation within the industry?

Similar research performed by Jaktman in 1998 showed that organizational values have an impact on the high-level architecture of software and should therefore not be neglected [3]. A survey of 40 Chinese companies performed by Leung in 2001 shows that quality is the last concern for management, after functionally correct, within budget and on schedule. Similarly, among quality characteristics, maintainability trails behind reliability and functionally correct, with only 35% of managers considering quality to be a key issue [2]. Software engineers want to produce quality software, but it seems that many organizations do not take the proper approach to reach this goal.

Two observation sessions conducted by Allison in 2010 [8] showed the following:

- **Case 1**: When a conflict arose between Agile teams within a non-Agile organization, the will of the organization prevailed and the Agile principles more or less disappeared.
- **Case 2**: Constant pressure from an organization on one of its development teams forced them to change their practices, despite significant resistance on the team's part.

Allison's conclusion is that the influence of the organization on the team is larger than the influence of the team on the organization.

It can therefore be surmised that organizational factors may impact product quality, as the values of the organization will influence how team decisions are made.

The objective of this observational study is to bring more evidence to existing literature on specific issues related to organizational values or structures, and software quality. The following sections present how each issue could be related to a generic case, and present the literature pertaining to this case, when it is available. We also suggest corrective action which, although case-specific, might prove useful for other practitioners facing similar issues.

A. *The generic case of "documenting complexity"*

The "documenting complexity" issue is related to knowledge management processes, especially on how development teams transfer knowledge to support and maintenance. The observed developers had many questions

on the legacy software in their organization. These questions can be summarized into the two following queries:

- Why does this specific piece of software exist? What does it do? Who needs this software?
- How does it work? Which resources (database, internet ports, etc.) does it use?

These issues are well-known in the knowledge management field. The need to record both the "what" (inner workings) and the "why" (design rationale) of software has been well documented [9], [10].

In the observed case, the complete system is broken down into software items like modules, scripts, libraries, etc. The purpose of each software item within the complete system is often unknown, as few of them are documented. Developers need to know how the software item works, in order to maintain its functionality. But they especially want to know its purpose, in order to ensure that their changes do not break it. To paraphrase Ko et al., the ideal, cost-effective, documentation system would be "demand-driven" instead of exhaustive [6]. Ko et al. encourages face-to-face interactions with colleagues because the data are available on demand and it requires little effort to record, maintain and transmit.

An up-to-date list of experts within the organization for each software item could also be interesting for future development efforts [11], although it cannot be considered a panacea by itself [12].

B. The generic case of "internal dependencies"

The "internal dependencies" issue is related to the concurrent development of inter-dependent modules. This is a common issue when using third party libraries, an issue which is resolved by providing multiple version support of by designing flexible services [13]. See for example, the multiple PHP language libraries supported [14].

Supporting multiple versions is not always financially possible within private organizations, however. For the observed organization, the software developed is executed only once on the organization's servers. There is no need to support multiple versions, because only the latest version will be executed at any given time. This may result in some conflicts, as each team wants to push its latest version to the organization's server, causing rippling issues within other modules currently under development.

This issue has not been directly studied in the literature. At the technical level, it can be related to the design of evolving families of web-services [13]. This approach might be a hard-sell however for an organization where only one version of the software is executed.

At the management level, it can be related to inter-team negotiations within global software development contexts [15]. Team negotiations are still an emerging research subject, as Guo and Lim presented in 2007:

"Despite considerable investigation on negotiation support systems (NSS), such research is largely in dyadic (i.e., one-to-one) context which is challenged by the observations of business practices: negotiating teams often appear at the bargaining table." [15]

This case study confirms the suspicions of Guo and Lim that team negotiations often occur within business contexts, and that these negotiations can lead to conflicts.

As a corrective action, in the case of parallel development of inter-dependent software modules, we suggest setting up a negotiation table to settle conflicts between the development teams. Given the effort required to manage negotiations and track results, efforts should be focused on dependencies with significant impacts on each project's success. The use of a software tool like the one suggested by Guo and Lim [15] could also help mitigate the effort overhead of negotiation tables.

C. The generic case of "external dependencies"

The "external dependencies" issue is related to acquisition and reuse processes, especially on how to interact with third party developers. Within the observed organization, the impact of contractual obligations with third party developers is twofold:

- First, the many different entities and contracts are an important knowledge hurdle for new developers, who need to know who to contact to obtain information, and which modifications are possible under the current contract.
- Second, the cost (in time and money) of CRs forces developers to make cost-conscious decisions, instead of quality-conscious ones.

Research on third party acquisition and reuse is abundant: For example, it is the purpose of the CMMI-ACQ [16] and it is covered by the Acquisition process of ISO 12207 [17]. The CMMI-ACQ, especially, outlines the need for training after the acquisition of third party software, in order to avoid the first impact presented above.

The second impact is also covered in the contractual negotiation sections of the two models. The importance of selecting the right third party developers cannot be underestimated: it is important to ensure that third party developers make the requested changes in a cost-effective and timely manner, and that they follow the rules of the trade. In the case observed, many CRs were performed either late or poorly, and did not meet the specifications given. These constant costly delays forced the development team to change their design for the worse, in order to avoid dealing with CRs.

As a corrective action, we suggest that the organization make sure that the team responsible for third party software acquisitions is aware of the CMMI-ACQ or ISO12207 processes, and of known challenges during large-scale scale software acquisitions [18]. The impact of a poor contract can be serious: in some cases, the customization performed by the company in order to work with the third party software is so extensive that moving to another supplier can be prohibitively expensive. Some of these acquisitions will last many years, or even decades.

D. The generic case of the "cloud storage" issue

The "cloud storage" issue is similar to the previous one, but has to do with testers instead of developers. The issue

here is related to the absence of testers during contract negotiations: some of the specifications of the development team had not been met by the contract.

However, given the sensitivity of the matter (vulnerability testing), negotiations are bound to be difficult. The issue is that the whole system works as a chain, where an input passes within multiple subsystems before producing an output. Some of these subsystems within the chain are supported by third party entities. The observed organization wants to test for specific security vulnerability, but some of these third party entities are not warm to the idea of having one of their clients *literally* hacking their systems.

Cloud vulnerability testing is currently an emerging research area. Its importance is undeniable: following the theft of 40 million Target credit card accounts, the company was forced to testify before the U.S. Congress and had to fork over US$60 million in costs to mitigate the issue [19]. The observed organization wanted to avoid a similar catastrophe, which could result in its customers' private information being made available on the Web or elsewhere.

Our suggested corrective action is to make sure that all relevant stakeholders (developers, testers, support, etc.) are involved when negotiating the acquisition of a software component with a third party. The development team should also make sure that all the activities planned for the project are possible within the current contractual obligations. For example, if the entire project hinges on a functionality that a third party entity cannot or will not deliver, the whole project can be thrown into jeopardy.

E. The generic case of "organically grown processes"

The issue of "organically grown processes" shows that processes typically appear on an as-needed basis, and do not follow a global plan. Allison and Merali describe the appearance of processes in the following way: *"Software processes can be considered to emerge by means of a structuring process between the context and the content of the action"* [8]. When the context and content of a development activity warrants it, a process element is added. An example would be an unexpected crisis, like a software patch causing a whole system to crash. Crises like these are usually followed by the introduction of process elements dedicated to preventing similar crises.

This issue highlights the importance of building an organization-wide process plan, such as the ones promoted by CMMI-DEV [20] or ISO 15504 [21]. The problem with organically grown processes is that they create islands of formality—zones where various software engineering processes have few interactions between them. Our suggested corrective action is to plan an organization-wide reviews aimed at detecting isolated processes and promoting information flows between processes. Project post-mortems could also serve to detect where missing information caused a problem, and whether someone in the organization had this information and could have prevented the problem.

To avoid resistance to change from software developers [22], process improvement should focus first on a better use of existing activities and materials. Changes to existing processes should also be gradual, with an emphasis on evaluating and demonstrating whether the change is useful.

F. The generic case of "budget protection"

Unfortunately, many organizations see only short-term benefits. They want correct functionality, within budget and on schedule, with good quality, in that order [2]. Reducing the cost of future maintenance is rarely a priority. This observational study provides more confirmation that working software is more sought after than quality software.

One of the causes of this problem is that budgets are closely monitored. Development teams are discouraged from performing costly corrections that would benefit software modules from other teams. These types of general-purpose corrections or improvements need to be performed in a distinct project, which can be difficult to get approved since there are no immediately countable benefits. Developers are therefore encouraged to apply quick patches, instead of solving the issue once and for all.

The impact of these quick patches has been widely documented and identified as "software aging" [23] or "technical debt" [24]. Technical debt is defined as a technical shortcut which is cost effective in the short term but expensive in the long term. Developers relying on patches during development can accelerate the accretion of technical debt. Quick patches applied haphazardly over other patches are certainly more prone to create errors than a careful resolution of the problem at hand.

Our suggested corrective action is to plan "special" budget items to support long-lasting corrections or corrections that are likely to benefit many modules. This special budget could also benefit inter-team negotiations. It might be easier to reach a settlement if the issue can be resolved in a separate special project.

This case presents the need to reach a compromise between developers who want to write perfect code, and managers who want to avoid "gold plating". A global view of how the organization's legacy code should be could help the development teams know where accrued technical debt might do more damage. Inter-team negotiations could be easier if priorities are set in a global software plan.

G. The generic case of "scope protection"

The "scope protection" issue is strongly related to deadlines. In the observed organization, changes in project scope are usually supported by an extra budget. However, budget changes do not necessarily change the project timeline.

Therefore, this is a direct application of Brooks' law, which states that adding new resources to a late development project only causes it to finish later [25]. In the case observed, the extra resources were in the form of developers added late in the process. These developers stood mostly idle for some weeks as their mentors were already swamped with work and could not train them properly.

Given the size of the project, it took a long time for the additional developers to perform a significant amount of work, as it required an important amount of time to acquire all the relevant knowledge. The addition of new developers

therefore slowed down the development process, as the current developers spent part of their time assisting the newcomers. For example, one new developer ended up doing secretary work during the deployment phase (manning the phone, taking notes in meetings, etc.) since he was not familiar enough with the code to modify it.

Projects with strict deadlines are risky, and should be carefully monitored to avoid last-minute unplanned activities. As Brooks wrote: *"A project is one year late one day at the time!"* [25]. The problem is that for each task added to an already tight deadline, the quality of the work takes a hit. Our suggested corrective action is either to allow deadline slippages, to enforce scope freezes on development projects, or to implement a more open scope negotiation process between teams involving a neutral judge (either a senior developer or the IT department manager).

H. The generic case of "organizational politics"

Another, more human issue is related to "organizational politics". The company's culture encourages managers to work outside the software process and rely on internal politics. The validation interview confirmed that observation. Results are not obtained solely by following due process, but also by calling in favors from colleagues.

This management approach has been labeled "organizational politics", because it uses a political approach akin to lobbying [26]. Previous research on this issue has shown that organizational politics are important in some organizations, ensuring that software development teams have the resources they need [26].

This is interesting because it shows that a CMMI or ISO 15504 certified organizational process is not sufficient to increase project success, as the organizational culture encourages employees to find solutions outside the prescribed process. On the one hand, processes can define useful information flows, but on the other hand humans prefer to work face-to-face rather than through documents and forms. Team members should therefore maintain a careful balance between formal development processes and informal human interactions, or as Allison and Merali put it, they must make some "negotiated changes" to the development processes [8].

I. The generic case of "human resource planning"

Many studies in software engineering pertain to the so-called "truck number", a metric attributed to Jim Coplien: *"How many or few people would have to be hit by a truck (or quit) before the project is incapacitated?"* ([7], p41). For the team studied here, the "truck number" is a real issue, as it is about to lose its only two members familiar with the initial project and specifications.

The development was performed mostly in silos, resulting in only one or two software developers gaining expertise in any given domain, which caused problems throughout the project. Whenever a developer took time off, most work on his/her area of expertise had to be postponed because no one else knew enough about how each piece of the puzzle worked. Similarly, when a developer got overwhelmed with work, the others could not help because no one had the appropriate knowledge.

The "truck number" risk is therefore not only a threat to project survival, but can also create delays when the knowledgeable developer becomes a bottleneck.

Team members should make sure that knowledge is appropriately distributed amongst them. Our suggested corrective action is to promote practices which promote knowledge sharing (e.g. pair programming, code reviews). The objective is to break the silos and have the developers work outside their areas of expertise once in a while.

J. The generic case of "undue pressure"

The "undue pressure" issue refers to priorities imposed by a higher authority, whether upper management, a client, or a respected colleague. However, this higher authority might not have all the knowledge required to impose these priorities. The higher authority dictates priorities according to what he/she perceives as important, and not necessarily what is really important at the team level. This can result in the developers working on something that does not warrant immediate attention, and thus wasting project resources.

The important role of upper management support for the introduction of new practices (e.g. software process improvement) has been confirmed many times in the past. There does not seem to be much research on the relevance of upper management decisions, however, nor on whether these decisions are coherent with the needs of software development teams.

Any intrusion into team dynamics by outsiders should be done very carefully. These intrusions can be useful to correct a problem, such as developers failing to apply a procedure [8]. Our suggested corrective action is to make sure the outsider has all the relevant information before intruding at the team level.

V. CONCLUSION

The objective of this paper is to provide accurate empirical data on the state of software engineering in practice in a professional environment. The list of issues presented in this paper is by no means exhaustive: many more minor issues were observed during the ten month study. Additionally, generalization of the issues to generic cases shows that the main issues presented here are not new; most of them have been previously discussed in the literature. What this paper is stressing is that any software development project should present few of these issues and ideally none of them.

While these issues might not affect project success, our observation shows that they do affect software quality. And quality factors can have a major impact on maintenance costs. If the Module developed in this project is successfully deployed, it will likely be used for the decades to come. The design flaws introduced because of the organizational issues presented here will no doubt come back to haunt at least a generation of developers to come, as the code written today will be tomorrow's legacy code. Is this the kind of legacy we want to leave for future software developers?

ACKNOWLEDGMENT

This research would not have been possible without the agreement of the company in which it was conducted, (which prefers to stay anonymous), and without the generous participation and patience of the software development team members from whom the data were collected. To all these people, we extend our grateful thanks.

REFERENCES

[1] J. Mathew, "The relationship of organisational culture with productivity and quality. A study of Indian software organisations," vol. 29, pp. 677-95, 2007.

[2] H. Leung, "Organizational factors for successful management of software development," Journal of Comp. Info. Systems, vol. 42, pp. 26-37, 2001.

[3] C. B. Jaktman, "The influence of organisational factors on the success and quality of a product-line architecture," Proceedings of ASWEC '98: Aus. Software Engineering Conference, Los Alamitos, CA, USA, 1998, pp. 2-11.

[4] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: data collection techniques for software field studies," Empirical Software Engineering, vol. 10, pp. 311-41, 2005.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," Proceedings of the 28th International Conference on Software Engineering (ICSE '06), Shanghai, China, 2006, pp. 492-501.

[6] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, United States, 2007, pp. 344-353.

[7] L. Williams and R. Kessler, Pair Programming Illuminated. Boston: Addison-Wesley, 2003.

[8] I. Allison, "Organizational factors shaping software process improvement in small-medium sized software teams: A multi-case analysis," Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010), Porto, Portugal, 2010, pp. 418-423.

[9] J. E. Burge and D. C. Brown, "Software engineering using RATionale," Journal of Systems & Software, vol. 81, pp. 395-413, 2008.

[10] P. Avgeriou, P. Kruchten, P. Lago, P. Grisham, and D. Perry, "Sharing and reusing architectural knowledge - Architecture, rationale, and design intent," Proceeding of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, United States, 2007, pp. 109-110.

[11] Y. Ye, "Supporting software development as knowledge-intensive and collaborative activity," Proceedings of the 2nd International Workshop on Interdisciplinary Software Engineering Research (WISER '06), Shanghai, China, 2006, pp. 15-21.

[12] D. W. McDonald and M. S. Ackerman, "Just talk to me: A field study of expertise location," Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 315-324, 1998.

[13] S. R. Ponnekanti and A. Fox, "Interoperability among independently evolving Web services," in Middleware 2004. ACM/IFIP/USENIX International Middleware Conference. Proceedings, Berlin, Germany, 2004, pp. 331-51.

[14] T. P. Group. (2014, August 20th). PHP: Downloads. Available: http://ca1.php.net/downloads.php

[15] G. Xiaojia and J. Lim, "Negotiation support systems and team negotiations: the coalition formation perspective," Information and Software Technology, vol. 49, pp. 1121-7, 2007.

[16] Software Engineering Institute, "CMMI for Acquisition, Version 1.3," Carnegie Mellon University, 2010, p. 438.

[17] IEEE Computer Society, "ISO 12207:2008 - Systems and software engineering — Software life cycle processes," ISO/IEC, 2008.

[18] A. Al Bar, V. Basili, W. Al Jedaibi, and A. J. Chaudhry, "An Analysis of the Contracting Process for an ERP System " Proceedings of the Second International Conference on Advanced Information Technologies and Applications (ICAITA-2013), S. Vaidyanathan and D. Nagamalai, Eds., CS & IT-CSCP, 2013, pp. 217-228.

[19] M. Riley, B. Elgin, D. Lawrence, and C. Matlack, "Missed Alarms and 40 Million Stolen Credit Card Numbers: How Target Blew It," Bloomsberg Businessweek, 2014.

[20] Software Engineering Institute, "CMMI for Development, Version 1.3," Carnegie Mellon Nuiversity, 2010, p. 482.

[21] IEEE Computer Society, "ISO/IEC 15504:2004 - Software Process Improvement and Capability Determination (SPICE)," ISO/IEC, 2004.

[22] M. Lavallee and P. N. Robillard, "The impacts of software process improvement on developers: A systematic review," Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012, pp. 113-122.

[23] D. L. Parnas, "Software aging," Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 1994, pp. 279-287.

[24] S. McConnell, "Managing Technical Debt," Construx Software Builders, Inc.2013.

[25] F. P. Brooks, The Mythical Man Month: Essays on Software Engineering: Addison-Wesley, 1975.

[26] A. Magazinius and R. Feldt, "Exploring the human and organizational aspects of software cost estimation," Proceedings of the 2010 ICSE Workshop on Coop. and Human Aspects of Soft. Eng. (CHASE 2010), Cape Town, South Africa, 2010, pp. 92-95.