HAL
archives-ouvertes.fr

Improving practices in a medium french company: First step

Mahugnon Honore Houekpetodji, Nicolas Anquetil

To cite this version:

Mahugnon Honore Houekpetodji, Nicolas Anquetil. Improving practices in a medium french company: First step. [Rapport de recherche] RMod. 2019. hal-02978015

HAL Id: hal-02978015
https://hal.inria.fr/hal-02978015

Submitted on 26 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving practices in a medium french company: First step

HOUEKPETODJI Mahugnon Honore*†, Nicolas Anquetil‡
*University of Lille, France
†Inria Lille - Nord Europe, France
homahugnon@gmail.com, nicolas.anquetil@inria.fr

***Abstract*—Legacy systems are old software that still does useful tasks. In industrial software companies, legacy systems are often crucial for the company business model and represent a long-term business investment. Legacy systems are known to be hard to maintain. This is the case in a french company whose main product is twenty years old software written in PowerBuilder. Our long-term goal is to help it re-engineer this system. But how to validate our intervention? Little data is available on the system and specifically, past versions of the source code are not easy to recover. This constrained us on the metrics we could use. In this paper, we present a lightweight model to characterize the situation of the system and allow us to monitor it in the future.**

***Index Terms*—Legacy system, software quality model .**

I. Introduction

Software companies have little spare resources to allocate to software quality improvement or code defect removal. As a result, they often develop features in a hurry. As the system grows, it becomes harder and harder to maintain. To ensure the future of the system, and through it, of the company itself, some re-engineering actions might be necessary. The long period of growth and evolution of the system as well as staff evolution, lead to problems such as dead code, duplicate code, and obsolete documentation. The developers at the origin of the application are no longer present, so a large part of the knowledge (and this at different levels of granularity) is scattered or lost.

This is the situation at a medium French company, which main business product we are studying. The system is written in an old language (PowerBuilder) typically unknown to young programmers which translates in a difficulty to hire new developers. There is also a sense in the development team that the system suffers from architecture erosion and is difficult to maintain or evolve which translates into a steep learning curve for new developers. We are trying to help this company improve its software engineering practices and restructure the system to improve the situation.

In this paper, we are trying to answer the following question: How to validate our intervention and can we measure its concrete impact on the system and its evolution?

For the reasons explained above, we cannot completely rely on current developers knowledge of the system. The only usable information about the system is its non-versioned source code and a ticket database. Past literature often relied on historical data on the source code, for example cyclomatic complexity [Gill and Kemerer, 1991], or number of lines of code [Port and Taber, 2018]. In lack of a reliable software versioning system, we could not use these solutions.

We mine and analyse the ticket database to overcome our problem. This paper presents the results of this work.

This paper is structured as follows: we start with related work in section II, followed by the section III in which we present tickets database mining background. In the section IV we present our methodology and conclude in section VI.

II. Related Work

Recently, research on mining software repositories has received much attention as it attempts to understand software evolution [Zhang and Kim, 2010]. Software defect history and source code history are often used to monitor software maintenance or predict the system's quality. For instance, software defect prediction using software repositories has already been extensively discussed in the literature as witnessed by many literature surveys: [Catal and Diri, 2009, Hall, Beecham, Bowes, Gray, and Counsell, 2011, Hosseini, Turhan, and Gunarathna, 2017, Li, Shepperd, and Guo, 2019, Malhotra, 2015].

But these solutions rely on the availability of source code history. For example, [Gill and Kemerer, 1991] used McCabe's cyclomatic complexity metric divided by the size of the system to predict software maintenance productivity. [Port and Taber, 2018, Zhang and Kim, 2010] used software defect history and source code history to assess a system evolution to monitor maintenance.

In our case, the source code history is not easily available as the company does not make use of a version control system. Thus, we cannot use the methods presented above.

[Herzig, Just, and Zeller, 2013] propose to consider ticket database. He reports that 33.8% of the bugs studied among five open-source projects were misclassified. For this reason, other researchers try to automatically re-classify maintenance activities (e.g. [Levin and Yehudai, 2019, Mockus and Votta, 2000]).

At the company, maintenance activities classification is more reliable because it drives the entire software evolution process. We rely on this classification to make our analysis.

We develop a lightweight technique to analyse the ticket database.

III. Study background

In this paper, we are interested in characterising the system and its evolution to better evaluate the possible future impact of our intervention.

A. Presentation of the system studied

Legacy systems have a lifespan of several decades, decisions made at the beginning of development and their evolution over the life of the software are often lost. This is the case of the system we are studying which is more than 20 years old. It is written in Powerbuilder. Powerbuilder is a programming language and integrated development environment initially developed by PowerSoft. The first version was published in 1992. Powerbuilder application components are grouped in libraries. A Powerbuilder library contains differents type of objects: datawindow, user object, global function, menu, etc. Powerbuilder is a procedural language that evolved to (partial) object-orientedness: inheritance and object-oriented features are limited to some object types (windows, user objects and menus).

The system has 3 MLOC, in 117 Powerbuilder libraries. The largest library is over 300 KLOC. The development team varied over time but counts 15 people at the moment, without the testing team which is external to development.

PowerBuilder version supports version control only since its 2017 version, and even this supports is still not entirely satisfying, for example conflict resolution is performed automatically by the system by choosing one version over the other without asking the developer. For these reasons, version control management was never used on the system studied.However some old major versions or archived in separate directories. Versions before 2012 are completely lost. Some version control is done informally inside the team. When developers work on the same version of the system, they chat with each other to notify which part they are modifying so that others won't modify the same part. Also, each developer has to write a comment in the source code with his/her name and the date the modification he/she performed.

B. Ticket

At the company, tickets are stored in the tickets database since 2000. A ticket represents a unit work. The ticket database drives the entire software evolution process: assignment of work to developers, management of the workflow to answer a client request, billing information about each task. There are tickets for fixing defects, writing documentation, adding new features, etc. As a consequence, we can rely on a ticket's classification because it is reviewed by the team manager to ensure that it is properly handled.

At the company, a ticket includes the following characteristics among other things:

- the creation date
- the closing date
- the estimation of the time needed by the developer to work on the ticket
- time spent by a developer
  - time to analyze
  - time to implement a solution
  - time to test
- the library(ies) impacted

IV. Methodology

Our long-term goal is to help developers re-engineer the system at a small cost. This is needed to improve the general quality and make future evolution easier, but it needs to be cheap to convince upper management investing in this purely technical task.

Therefore, we need to validate our intervention as well as the restructuring work developers might engage in the future.

A. Souce code versionning

Our first action was to introduce a version management system. We chose Subversion (SVN) because it was simpler to explain to developers that were not familiar with the concepts of version management. We gave a short training session to the developers, and they are now starting to use it as part of the new process.

Besides, we are reconstructing a summarized source history in SVN by importing one after the other the different major versions currently stored in directories (versions 2012 to 2019). We hope that this will help us analyze the system's historical code changes to have additional information about the system evolution.

B. Ticket data analysis

Our goal was to characterize the system to be able to measure our impact and that of the developers in the future. The problem was to extract information from the system that was relevant for the company and for which we could hope to have some impact. All this within the very restricted context described above.

We turned to the ticket database and elected the following questions:

- What is the proportion of evolution versus correction tickets?
- How long does it take to close a ticket?
- Development time spent on a ticket?
- Testing time spent by the developer on a ticket?
- Testing time spent by the tester (other teams) on a ticket?
- Difference between the time estimated and the actual time for a ticket?

The ticket database first needed some cleaning:

- Tickets may concern several software systems in the company and different development teams. We thus selected tickets related to the system we are studying.
- The ticket database contains data from 1998 until today. Before 2004, tickets information are not always consistent, for example, missing creation date or missing/incorrect category. This constrained us to eliminate tickets prior to 2004.

- Up to recently, the name of the library concerned by a ticket was filled manually (free text). As a consequence, there are some typos in the names that need to be corrected. A simple and frequent case is two swapped characters *cwm_liq_ou* instead of *cwm_liq_uo* ("uo" PowerBuilder User Object). Some more complex cases require an understanding of the application domain and the system, for example, *uo_liq* instead of *cwm_liq_uo*. We identified all the errors manually and corrected them.
- The category of the ticket is fine-grained whereas we are only interested in evolution versus corrective tickets. In our analysis, we recategorized the tickets according to some simple dictionary provided by the team manager.
- Durations are registered in a free text format with some conventions. As should be expected, the conventions were not always respected (e.g. "5d" or "5days") and there were typos. We created a simple converter which covers all the cases and converts them into numerical values.

For all the questions on time (for example time to close the tickets), the tickets are grouped by month of creation and their value are averaged for the month of creation. We compute the data for our questions as follows:

- We compute the *closing time* of a ticket simply as the time between creation and closing. This is only computed for closed tickets, the other ones are ignored.
- The time spent by the developers to implement the solution for a ticket is recorded in a "Time" table related to the ticket. We sum all "Time" rows linked to the ticket that are marked as development.
- The time spent by the developers to implement the solution for a ticket is recorded in a "Time" table, we sum all "Time" rows linked to the ticket that are marked as testing
- In the ticket there is a field recording the estimated time to solve it. This field is filled by the team manager. There is also a field recording the total time spent on the ticket by the developer. We compute the difference between the total time spent and the estimated time. If this difference is positive, there was under estimation, and if it is negative there was over estimation.

We finally plotted the data on a 2 dimension graph. We noticed a very large variation of the results from month to month (nonstationary results). This made it hard to interpret the graphs. To get smoother plots, we computed the moving average of [month - 2; month + 2] for each month. So for example, plotted data for July 2010 corresponds to the average time of all tickets opened between May 2010 (inclusive) and September 2010 (inclusive). We will also plot the linear regression of the curve for each graph to get a better notion of the trend.

V. Results

A. Souce code versionning

The introduction of SVN only started one month ago and we do not have yet significant results to report.

For the reconstruction of the summarized source history, we already imported major versions sorted by release date until 2015 in an SVN repository. At the end of the importing, we could analyze changes between release at the source code level.

B. Data collection

TABLE I
Ticket analysed

Tickets	Defect	Evolution
27380	15407 (56%)	11973 (44%)

Table I gives the number of defect versus evolution tickets. We note that 56% of defect tickets seems a high proportion. Literature (e.g. [Pigoski, 1996]) states that the proportion is typically 20% to 25% of defect tickets. This is an issue that we will investigate and monitor in the future.

C. Time to close a ticket

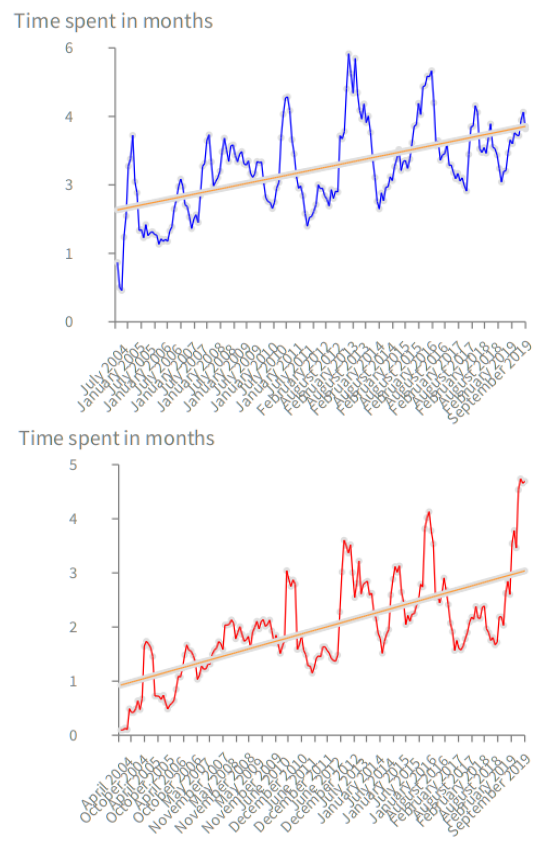

Fig. 1. Time to close evolution (up) and defect (down) tickets

Figure 1 presents the values of closing time for evolution (up) and defect (down) tickets. One can notice a large variation of plotted time, even after smoothing it with the moving average. We don't see it as pointing to a significant problem with the system.

It takes longer (on average) to close an evolution ticket than a defect ticket which seems natural.

We also note that the time is augmenting over the years for both categories which may be a sign of declining quality. Finally, we note that the closing time over the 15 years was multiplied by 2 for evolution tickets and 3 for defect tickets. Again this is an issue to investigate.

D. Developement time spent on a ticket

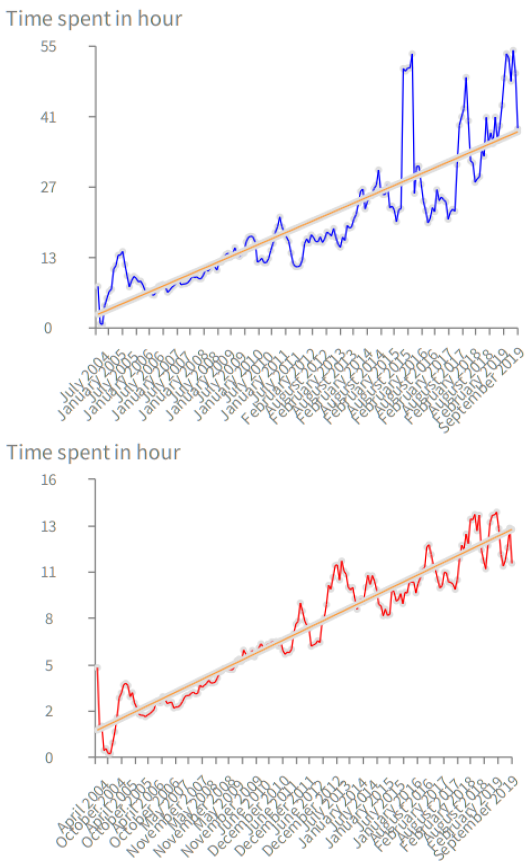

Fig. 2. Time spent by developers to implement his solution for evolution (up) and defect (down) tickets

Figure 2 presents the time spent by developers to implement a solution to evolution (up) and defect (down) tickets. As Figure 1, Figure 2 shows a large variation even after smoothing.

It takes longer (on average) to implement a solution for an evolution ticket than a defect ticket. This seems natural.

We also note that the time is augmenting over the years for both categories which may be a sign of declining quality. Finally, we note that the time spent by developers to implement a solution was multiplied by a similar factor for evolution and defects (about 8 or 10).

E. Testing time by the developper

Figure 3 presents the trend of the time spent by developers to test their solution for evolution (up) and defect (down) tickets. We can observe that the time is decreasing for evolution tickets and stable for defect tickets. This might be due to the

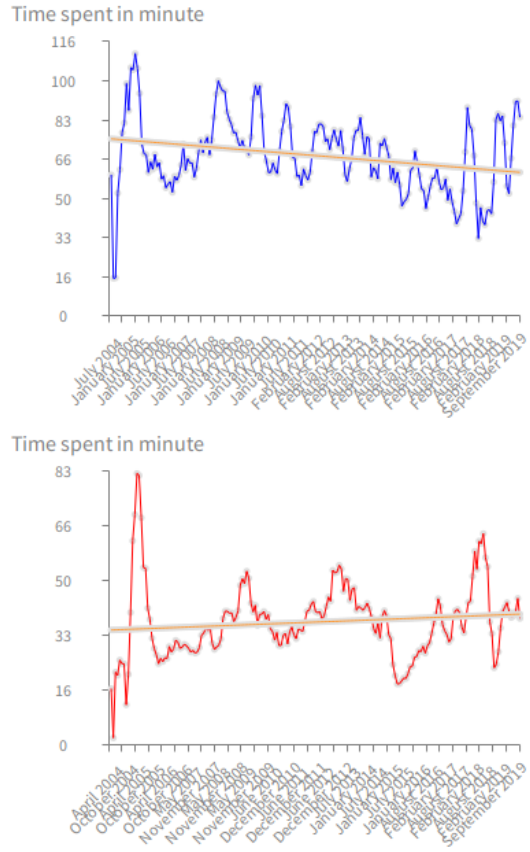

Fig. 3. Time spent by a developer to test his solution for evolution (up) and defect (down) tickets

pressure to deliver new evolutions. We will come back to this in the next section. This declining testing time may result in a loss of quality of the system that could explain the results shown in the previous sections.

F. Time estimation

Figure 4 shows the difference between the time spent by the developer and the time estimated for evolution (up) and defect (down) tickets. We can observe two obvious periods: before and after May 2013. Before May 2013 the time spent by the developer is superior to the time estimated with and increasing difference for both evolution and defect tickets. This means that the developers did not have enough time to solve the tickets. It constrained developers in their task which might have impacted negatively the quality of the solutions and therefore of the code.

After May 2013 the time estimated is superior to the time spent for evolution and defect tickets. This estimate is improving for evolution tickets as the regression line is closing to 0. For defects tickets, the overestimation tends to increase. We could relate this to diminishing testing time (Section V-E) for evolution ticket (that could be due to a lack of time) whereas the testing time remains constant for defect tickets which still enjoy a clear over-estimation. However, it must be noted that we did not see the same two periods difference

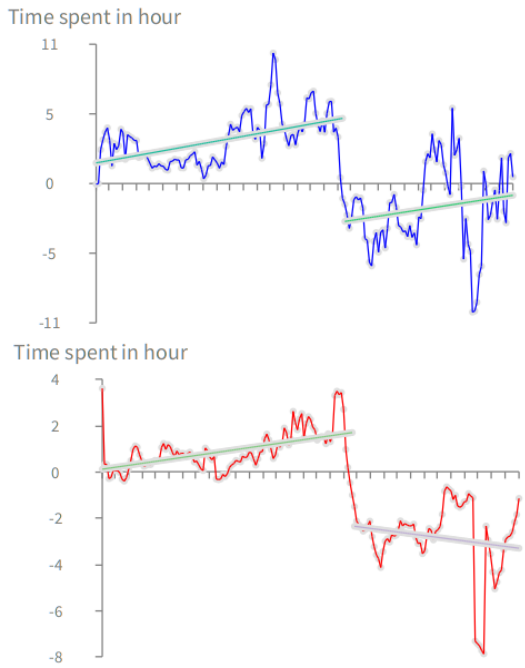

Fig. 4. Difference between time spent by developers and time estimated for evolution (up) and defect (down) tickets

in the previous section, therefore it is not clear that under-estimation of the time is at the source of the diminishing testing time.

VI. Summary and Conclusions

In this paper, we present our first steps in improving practices in a medium company. We first introduced a version control system in the evolution process. We chose Subversion as it seemed simpler to present to developers that had no previous knowledge of versioning. The development team is now using it and the source code is now versioned. This will open the door for further analyses specifically on the code repository to understand the system evolution.

As we want to monitor our future actions on the system, we analysed the ticket database to characterize the current situation. The results seem to point to the declining quality of the system (it takes longer to implements tickets), but we also noted an improvement in the estimation of the workload of a ticket.

References

Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Syst. Appl.*, 36(4):7346–7354, May 2009. ISSN 0957-4174. doi: 10.1016/j.eswa.2008.10.027. URL http://dx.doi.org/10.1016/j.eswa.2008.10.027.

Geoffrey K Gill and Chris F Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284, 1991.

Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486840.

Rebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering*, PP:1–1, 11 2017. doi: 10.1109/TSE.2017.2770124.

Stanislav Levin and Amiram Yehudai. Towards software analytics: Modeling maintenance activities, 2019.

Ning Li, Martin Shepperd, and Yuchen Guo. A systematic review of unsupervised learning techniques for software defect prediction. *arXiv preprint arXiv:1907.12027*, 2019.

Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504 – 518, 2015. ISSN 1568-4946. doi: https://doi.org/10.1016/j.asoc.2014.11.023. URL http://www.sciencedirect.com/science/article/pii/S1568494614005857.

Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings International Conference on Software Maintenance ICSM-94*, pages 120–130, Victoria, BC, Canada, 2000. IEEE Comput. Soc. Press. ISBN 978-0-8186-6330-7. doi: 10.1109/ICSM.2000.883028. URL http://ieeexplore.ieee.org/document/883028/.

Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996. ISBN 0471170011, 9780471170013.

Dan Port and Bill Taber. An empirical study of process policies and metrics to manage productivity and quality for maintenance of critical software systems at the jet propulsion laboratory. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 37. ACM, 2018.

Hongyu Zhang and Sunghun Kim. Monitoring software quality evolution for defects. *IEEE Software*, 27:58–64, 07 2010. doi: 10.1109/MS.2010.66.