Visualizing and Managing Technical Debt in Agile Development: An Experience Report

Paulo Sérgio Medeiros dos Santos[1], Amanda Varella[2], Cristine Ribeiro Dantas[2], and Daniel Beltrão Borges[2]

[1] Federal University of Rio de Janeiro, System Engineering and Computer Science Department, Cidade Universitária – Centro de Tecnologia.
Rio de Janeiro, Brazil
pasemes@cos.ufrj.br
[2] Petrobras, Exploitation and Production Business Solutions, Centro
20031-912 Rio de Janeiro, Brazil
{amanda.varella,cristine.dantas,
daniel.borges}@petrobras.com.br

Abstract. This paper reports the experience of an architecture team of a software development department with 25 agile teams in supporting technical decisions regarding technical practices. The main motivation to use technical debt metaphor was its acknowledged potential in driving software development and maintenance decisions, especially those long term maintenance tradeoffs which are usually less visible to developers and decision makers in general. We propose the use of a "technical debt board" with main technical debt categories to manage and visualize the high-level debt, combined with tools to measure it at low-level (software metrics and other kind of static analysis). We have found that our approach improved the teams' awareness about the technical debt, stimulated a beneficial competition between teams towards the debt payment and enhanced the communication regarding technical decisions.

Keywords: technical debt, software quality, visualization, agile practices.

1 Introduction

One of the main tenets that make agile methods effective is the right balance between the importance given to the people developing the software and the engineering practices dedicated to keep its quality. In establishing this balance, agile teams can leverage from the embodied tacit knowledge in the team and the technical readiness for change of the software product [1] to attain its primary objective: deliver value.

However, this equilibrium can be hard to achieve. And, if not consciously monitored, it seems that it can be more easily inclined towards the people or management side. The following aspects help explain this situation. First, the agile development de-emphasis to long-term planning in favor of short-term adaptiveness, although it represents a strength in a rapidly changing development environment, can create a temptation to neglect best practices that are essential to long-term success [2]. Second, most major agile methods such as Scrum and Crystal are more focused in the

H. Baumeister and B. Weber (Eds.): XP 2013, LNBIP 149, pp. 121–134, 2013.

managerial aspects of software development than in providing engineering guidance [3] – one important exception is Extreme Programming. Last, the engineering practices commonly used in agile methods require highly qualified professionals [4] which must be able to deal with the lack of upfront design and investment in the life cycle architecture [1, 5] besides, these professionals must be capable of realizing automated testing and continuous integration [6].

In fact, all these aspects are implicitly present in the agile manifesto (http://agilemanifesto.org/). We cite four principles directly related to this discussion: (i) the continuous delivery of valuable software to the client, (ii) continuous attention to technical excellence and good design enhances agility, (iii) simplicity – the art of maximizing the amount of work not done – is essential and (iv) welcome changing requirements, even late in development. Combining the ideas of these principles, we have the following challenge: how to balance quality, simplicity, agility and welcome change in delivering value to the client?

Thus, although delivering value is the ultimate objective of agile methods, delivering it as fast as possible without an adequate attention to the engineering practices can represent a problem. In 1992 Ward Cunningham created a metaphor to a code that is written in a fast and "dirty" way or, more technically, code that is produced taking shortcuts that fall short of best practices. He called this metaphor Technical Debt [7]. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that has to be done in future development because of inappropriate design choices [8]. This includes all aspects of software development including its documentation, test cases and source code.

There are many reasons to get into technical debt – not all bad, especially when it is taken in a conscious manner. In addition, technical debt is not limited to practices and techniques associated with the code design itself. More broadly, technical debt can be characterized by aspects associated with the development of the software product as a whole [9], including: lack of automated tests, unnecessary coupled code, duplicated code, infrastructure related issues like flawed automatic building, lack of continuous integration and automated deployment.

Although the issues related with the technical debt are in the surface technical, this type of issues cannot always be objectively addressed. It is not possible to pay the technical debt simply saying to the developers: you must write automated tests, don't couple your code, don't duplicate your code or refactor your code. When dealing with this kind of situation, cultural factors must also be managed in order to make developers capable of dealing with tension between engineering best practices and other factors such as ship date and skills of engineers that are available.

This paper reports an experience of how a software development department of a big oil company located in Brazil dealt with a scenario similar to the one described above. In an advanced stage of agile adoption, the department was facing a condition where the managerial aspects were already fairly consolidated with the introduction of Scrum, but the engineering practices were lagging behind in terms of maturity. We describe how we have used the technical debt metaphor to stimulate software developers to bring the managerial/engineering equilibrium to an optimal state where the value delivery is maximized in long-term.

2 Background

As an oil and gas company, Petrobras (http://www.petrobras.com.br) develops software in areas which demands increasingly innovative solutions in short time intervals. The company started officially with Scrum in March of 2009, using its lightweight framework to create collaborative self-organizing teams that could effectively deliver products. After the first team had adopted Scrum with a relative success, the manager noticed that the framework could be used in other teams, and thus he invested in training and coaching so that the teams could also have the opportunity to try the methodology. At that time, only the software development department for E&P, whose software helps to Exploit and Produce oil and gas, had management endorsement in adopting scrum and agile practices that would let teams deliver better products faster. About one year and half later, all teams in the department were using Scrum as its software development process. The developers and the stakeholders in general noticed expressive gains with the adoption of Scrum. The results had varied from the skill of the team leadership in agile methodologies, customer participation, level of collaboration between team members, technical expertise among other factors.

The architecture team of the software development department for E&P was composed of four employees, whose responsibility was to help teams and offer support for resolution of problems related to agile methods and architecture. At that time, it had to work with 25 teams which had autonomy regarding its technical decisions. In fact, autonomy was one of the main managerial concerns when adopting agile methods.

After Scrum adoption, there was active debate, training and architectural meetings about whether Agile engineering practices should also be adopted in parallel with managerial practices; in hindsight, it would have accelerated the benefits had they been adopted. But the constraints of time and budget, decisions made by non-technical staff, and the bureaucracy in areas such as infrastructure and database, led to the postponement of those efforts initially. Moreover, the infrastructure area had only build and continuous integration (CI) tools available. And, unfortunately, these tools were not taken seriously by the teams. The automated deployment was relatively new and was postponed because of fear of implementing it in immature phase. Other tools and monitoring mechanisms were not used by the teams even so the architecture team was aware of its possible benefits.

Despite all initiatives in training and supporting in agile practices such as configuration management, automated tests and code analysis, teams, represented by 25 focal points in architectural meetings, did not show much interest in adopting many agile practices – particularly technical practices. Delivering the product on the date agreed with the customer and maintaining the legacy code were the most urgent issues. Analyzing retrospectively it seems that the main cause for this situation was that debt was getting accrued unconsciously. Serving the client was a much more visible and imperative goal. This can be one explanation for the ineffectiveness of prior attempts in introducing technical practices. Be it by the means of specialized training or by the support of the architecture team.

With these not so effective attempts to promote continuous improvement with teams, the architecture team sought a way to motivate them to experiment agile practices without a top-down "forced adoption". The technical debt metaphor, described in next section, was the basis for the approach.

3 Related Works and the State of the Practice

Technical debt has been a central theme among researchers and practitioners in the last years as an alternative perspective for software development and maintenance decisions. It offers a real world metaphor which is naturally understandable by most software stakeholders and serves as tool to evaluate the tradeoffs between proposed enhancements, corrective maintenance and technical/non-functional improvements. Besides that appeal, what seems to be the most significant contribution of the technical debt prism is that it brings to the light the long term maintenance tradeoffs which are usually less visible to developers and decision makers in general.

It is possible to identify three main effort directions in the technical literature and informal sources (blogs) related to technical debt.

The first are those [10, 11, 12, 13] seeking to identify the main properties of technical debt and conceptualize the main sources of its accumulation. This includes interviews with practitioners to see how they interpret technical debt and how it manifests in their daily activities [10, 13]. In addition, it also includes discussions about technical debt characteristics, such as described in [11] and [12]: visibility (to make it visible for daily decisions), value (to estimate its size and help deciding when it should be paid) and intentionality (unintentional, when it is a result of low quality work and intentional, for tactical (short-term) and strategic (long-term) reasons).

The second group [14, 15, 16, 19] is linked to how the debt can be managed and deciding when it should paid. There are many approaches to do that. In [14] four of them are cited, including the simple cost-benefit analysis, where the cost of paying the interest versus principal is analyzed, and the portfolio-based, where technical debt items are treated as assets that composes a portfolio managed to maximize the return of investment or minimize the investment risk. Both [15] and [16] are cost-benefit approaches and [19] proposes a portfolio-based approach. Valuing and making technical debt visible are the basic inputs for these approaches. And, in fact, these two properties are directly related to each other as it only possible to manifest something that can be observable (in this case, valued). Examples of technical debt measurement ranges from the use of rough estimates [15, 19] to the more precise quantitative data based on source code metrics [16].

The use of source code or, generally, "low-level" software metrics to estimate a value for technical debt forms the third main active area. In [17], automatic static analysis is empirically evaluated as mean to quantify the values of technical debt at code level. And in [18] the relative technical debt value associated with three code smells (data class, duplicated code and god class) is investigated.

Despite the effort on identifying means of how technical debt can be measured (and visualized), it seems that only low level aspects are being focused on. The use of

static analysis tools and source code metrics are self-explanatory examples. And even on those works that describe how it can be estimated qualitatively, the attention is turned to software products and not to software practices. For instance, examples of technical debt items in [15] are: architectural design violations, test skipped, outdated documentation and design debt.

Given the context aforementioned, especially regarding the role of the architecture team serving various teams in parallel and the fact that the teams have autonomy in its technical decisions, there was a need for the technical debt estimation and visualization in a "macro-level", i.e., not only associated with source code aspects but with technical practices involving the product in general. This would give the opportunity to see the actual state of the department and indicate the "roadmap" for future interaction with the development teams.

The actions involved in introducing this kind of visualization and the management activities based on that visualization are described in next section.

4 Actions

Given the challenge in addressing the issues caused by the technical debt, the architecture team started to discuss some initiatives that could help the area: (i) recognize that the lack of attention paid to the technical debt was a problem (teams and management), (ii) visualize the existing technical debt, (iii) quantifying the amount of technical debt, (iv) create mechanisms of feedback to see the technical debt rising or decreasing and (v) take actions to correct implementations that lead to technical debt.

It is important to mention that these actions were not planned upfront, but they emerged according to the feedback that the architecture team was having in trying to implement the technical debt awareness in the department.

4.1 Recognize That the Lack of Attention Paid to the Technical Debt Was a Problem

The first step the architecture team had to take was to make sure every developer knew the concept of technical debt. As in the pair "reckless x inadvertent" in Martin Fowler's [8] quadrant of technical debt, most of the team members did not know the exact meaning of technical debt. The architecture team then started to do a series of presentations about the theme. What is technical debt; what is its size; why do we accumulate technical debt and how do we pay it; and how to benefit from technical debt were topics presented to the audience.

The architecture team had also the challenge to speak to different audiences. More technical presentations were made to the teams, but to the upper level management, another language was needed, so everyone could understand the topic by their own point of view.

At that time, many teams were already struggling with problems of poor architecture, rework, delays, and poor quality. All of these issues where impacting the relationship with their clients.

4.2 Visualize the Existing Technical Debt

The software development department was already having some initial Kanban [21] implementations. As one of the main principles of Kanban is visualization, these ideas were permeating the minds of the group, and many initiatives of change management were taking visualization into account.

The architecture team modeled a board, where the lines corresponds to teams and the columns are the categories and subcategories of technical debt, based on the work of Chris Sterling [9] as illustrated in Figure 1.

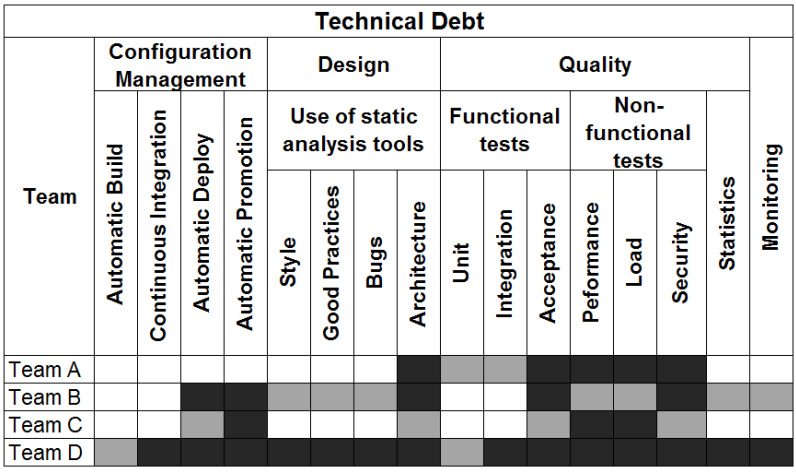

Fig. 1. Technical debt categories

In each cell, formed by the pair team x technical debt category, the maturity of the team was evaluated according to predefined criteria. Examples of these criteria are displayed in Table 1. For full description please see Appendix A. Notice that in the real board shown in Figure 2, we used the colors red, yellow and green to show the compliance level of each criterion. So, we kept the reference to these colors in the text even though Figure 1 uses a gray scale (white = green, yellow = light gray and red = dark gray).

The criteria just provided a direction of what kind of practices would be focused, but not give directives on how it could be achieved. This was the moment where the architecture team could offer its support. In addition, the criteria were not a rigid target. For instance, for unit tests, the ideal coverage level was dependent on the system architecture, technologies involved (e.g., programming language and frameworks), the criticality of the application and other factors. All of this was subject of discussion between the development teams and the architecture team. And that was one of the biggest benefits on bringing the debt visible.

The architecture team, in its internal conversations, was concerned that this approach could make the teams feel compelled to follow the orientations. This was not the objective, on the contrary the objective was to make teams aware of the

Table 1. Criteria examples for technical debt assessment

	Continuous Integration	**Unit Tests**
Red	There is no job in the CI Tool.	There are no Unit Tests.
Yellow	There is a job scheduled in the CI Tool.	There are some Unit Tests.
Green	There is a job scheduled in the CI Tool and the team is committed to keep the build working (compiling and with unit tests passing).	There are Unit tests in a level that the team is comfortable with.

problem and take their own actions to amend their technical difficulties. And if the teams were not capable of addressing the problem, the architecture team should be consulted for support. Thus, this initiative was first presented to the teams' focal points and it was explained that the main objective was not to constrain anyone, but to allow the visualization of their actual state regarding the technical debt and have the possibility to monitor their own progress. The proposal was presented in a very objective way, focusing on the engineering issues. The focal points did not offer resistance. Actually, many of them thought that the initiative was a good opportunity to improve their overall work (even with their own "problems" exposed).

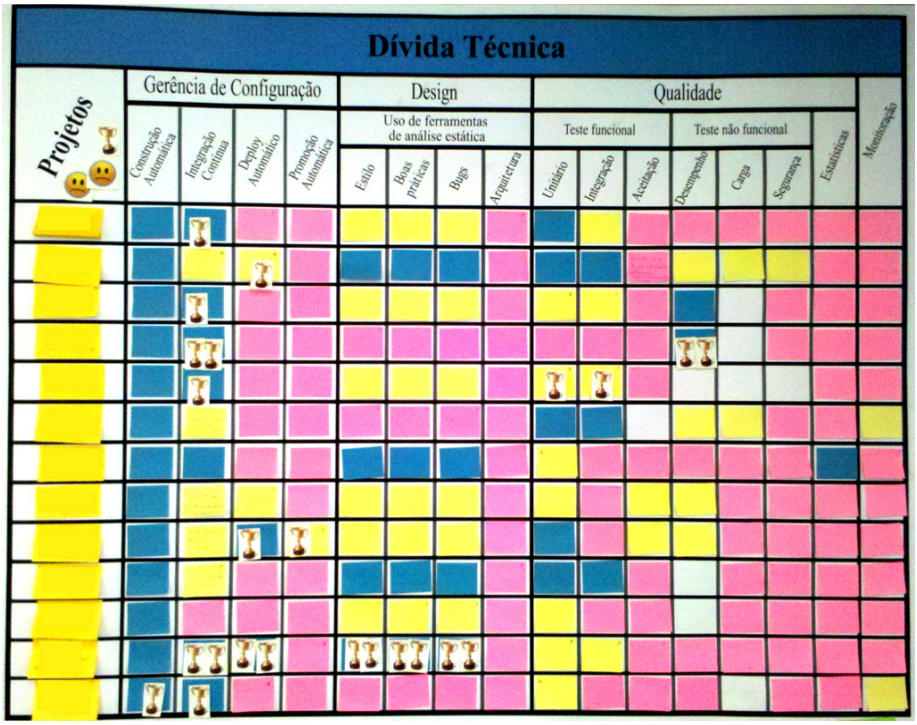

Fig. 2. Real technical debt board

After the design of the technical debt board, each team was invited to a rapid meeting in front of the board, where all team members talked about the status of each criterion, translating it to the respective color. During these meeting the teams could also conclude that some categories were not relevant or applicable for their systems. This meeting should happen every month, so that the progress of each category could be updated. At the end of the meeting, the team members agreed which of the categories would be the aim for the next meeting or, to put it another way, where they would invest their efforts in reducing the technical debt. The real board is presented in Figure 2. Blank cells represent categories not relevant or applicable. Team names from in the first column were removed from the figure.

4.3 Quantifying the Amount of Technical Debt

To measure the technical debt at source code level, the architecture team has made use of the tool Sonar (http://www.sonarsource.org/) [20]. Sonar has a plugin that allows estimating how much effort would be required to fix each debt of the project. Sonar considers as debts: cohesion and complexity metrics, duplications, lack of comments, coding rules violation, potential bugs and no unit tests or useless ones. The details of its formula can be found in [20]. The important aspect is that an estimative is calculated, and Sonar shows the results financially and the effort in man days necessary to take the debt to zero (the daily rate of the developer in the context of the project must be informed).

It is important to mention that Sonar, in fact, use many other tools internally to analyze the source code – each one for different aspects of the analysis. It works as an aggregator to display results of other tools such as PMD, Findbugs, Cobertura and Checkstyle among others.

4.4 Create Mechanisms of Feedback to See the Technical Debt Evolution

Having a visualization of the actual state of the technical debt and having it quantified was important step. However, having the debt quantified in a tool, and making some adjustments in the course of the system once a month would not be enough. The teams could make the debt rise during a whole month without even knowing about it.

To address this situation, the architecture team created a virtual tiled board (Figure 3), where each tile had information about the build state of each team in the department. The major information was the actual state of the build and the project name (which was removed from figure). If everything was ok (compilation and automated tests), the tile is green (white in Figure 3), if the compilation was broken, the tile turns red (dark gray in Figure 3) and if there were failed tests, the tile turns yellow (light gray in Figure 3). Besides the build information, there is other information: total number of tests, number of failed tests, test coverage, number of lines and technical debt (calculated in Sonar).

The virtual tiled board was placed in a big screen in a place where everybody in the room could see it from their workplaces. The main objective was that when the team members saw their failed build and that instant feedback would lead them to make corrective actions so the build could go green again.

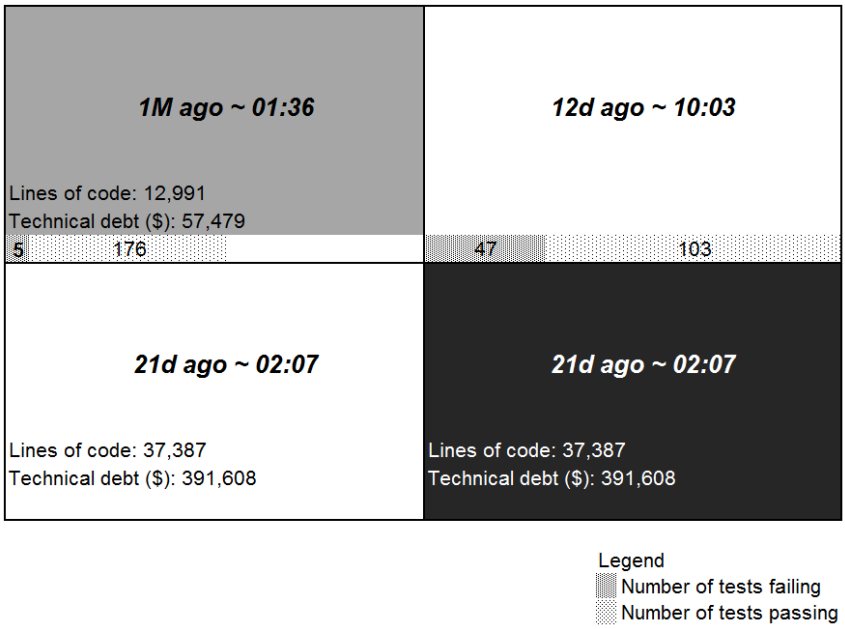

Fig. 3. Some of the virtual tiled board cells in detail (project names were removed from it)

4.5 Take Actions to Correct Implementations That Lead to Technical Debt Rising

As the mechanisms of feedback were implemented, the teams had instant information about what should be done to lower the levels of technical debt. With this information, they could prioritize which categories they would try to improve in the next month. If the team had some difficulties addressing any of the categories, they could call upon the architecture team support. In the following months after the board implementation, the architecture team kept making presentations about each category and how to deal with them.

5 Lessons Learned

In general terms, we think that the aforementioned actions can lead to small changes that over time will add up to significant positive change for teams and organizations. The main evidence for that, in our experience, is summarized below.

5.1 Make Visible. Don't Dictate.

The line between being firm about the value of implementing Agile practices and sensitive to the freedom and independence of teams is a difficult one to take right. With the board exposed, the approach was to encourage teams to self-evaluate theirs

technical debt, instead of someone, in our case the architecture team, pointing out problems. As a result, the teams showed initiative on seeking the architecture team for help with issues related to technical debt, as for instance, how to implement automated deployment or how to improve the source code testability for unit tests.

From the moment the debts were inserted, interest began to be contracted, and at some future time it may have to be paid. Making them visible and managing them, allowed strategic decision making to choose the best time to pay them – once it was possible to see how it was accumulating and its possible impacts/costs throughout the software development lifecycle.

Visualization was a powerful way to simplify complexity, expose the reality and, consequently, motivate teams to improve.

5.2 Improved Communication

Again, visibility was a key enabler to improve communication among development teams, architecture team and upper management. It turned the discussions around technical issues focused and oriented much efforts towards a common (visible) goal. The meetings around the board is now a regular practice in the department and in many situations development teams have opportunity to discuss about (the once unfamiliar) techniques to deal with their debt.

Another important benefit of the afore-described approach was establishing the basic concepts and tools around the technical debt theme which, again, facilitated the discussions. Developers are now more aware of the main factors that can contribute to the technical debt accumulation, are more open to discuss about it and know how to measure and address (technical practices) it.

5.3 Debts Paid at a Rate Higher than Expected

Besides the mentioned benefits, it was observed that debts were paid at a rate higher than expected. We interpreted this as a result of the competition among teams. In addition, we have noticed that this was stimulated by the introduction of gamification elements. Gamification [23] is the application of game elements and digital game design techniques to non-game problems, such as business and social impact challenges. It is used in applications and processes to improve user engagement, timeliness, and learning.

To apply gamification elements in our context items such as trophies exposed at the board of technical debt motivated teams to improve quality and pay debts. Every improvement made at the board, earned the team a trophy (the board in Figure 2 has some attached to it). This made visible how teams were evolving and kept the motivation for sustaining the progress.

We have calculated a raw estimative of how the technical debt payment progressed in the first year. To measure this progress, we kept the technical debt evolution history in a spread sheet, but only for the first thirteen projects (i.e., the first technical debt board) – this data was not made public. The progress was measured in the following manner. Supposing that the red/yellow/green represents an interval scale, the difference between red/yellow and yellow/green is one unit – for a max of 416

"units of debt" representing the worst situation of all projects "in red" in a board with thirteen projects. Considering the initial state of the board, the projects had 327 units of debt. From this initial state in 06-17-2011 to almost one year after in 05-14-2012, the projects already had 226 units. This constituted a progress of 30% which was above our initial expectations considering past experiences in fomenting technical improvements. It represented a great (visible) achievement in our department. It is interesting to notice, in addition, that this progress was not homogeneous among teams. Some teams evolved faster than others. And that, in our view, was one of the factors that stimulated competition.

The virtual tiled board also played an interesting role in bringing additional gamification elements to the technical debt management as teams immediately started to react to the red or yellow colors for broken builds. This was a result of an emerging social commitment of being seen different among their peers who kept their build green. The build status changes minutes after the source code check-in/commit by the team and this rapid feedback loop caused a strong change in the culture of the team members who after just a couple of weeks were already treating the build status with a high priority. This improved the perception of the teams in keeping their main/trunk branch closer to a deployable state as possible.

Thus, in addition to visibility, gamification was a powerful mechanism to motivate teams in monitor their technical debt.

6 Final Remarks

After Scrum adoption, the most visible symptoms of dysfunction in our software development department were related to agile engineering practices, where teams were accumulating a huge amount of technical debt. This paper showed how an architecture team at Petrobras has managed the technical debt in an agile context, seeking to reduce the high costs generated by debt issued. Working the change management iteratively, getting feedback for new actions, the intense use of visualization, the application of concrete measurements, and working together with the teams in a collaborative, not imposing manner, all that in context had proved to be powerful tools to obtain the desired results.

Another important contribution of this paper was proposing an approach for addressing the technical debt at a high-level. The proposed approach, besides the use of tools to estimate technical debt based on low-level source code metrics and reports, involves people to analyze the main contributing technical debt factors and plans the appropriate time to deal with it. In this manner, the board as a visual instrument has demonstrated to be useful in our context.

References

1. Boehm, B.: Get ready for agile methods, with care. Computer 35, 64–69 (2002)
2. Dinakar, K.: Agile development: overcoming a short-term focus in implementing best practices. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, pp. 579–588. ACM, New York (2009)

3. Abrahamsson, P., Warsta, J., Siponen, M.T., Ronkainen, J.: New directions on agile methods: a comparative analysis. In: Proceedings of the 25th International Conference on Software Engineering, pp. 244–254 (2003)
4. Merisalo-Rantanen, H., Tuunanen, T., Rossi, M.: Is Extreme Programming Just Old Wine in New Bottles. Journal of Database Management 16, 41–61 (2005)
5. Mishra, D., Mishra, A.: Complex software project development: agile methods adoption. Journal of Software Maintenance and Evolution: Research and Practice 23, 549–564 (2011)
6. Svensson, H., Host, M.: Introducing an Agile Process in a Software Maintenance and Evolution Organization. In: 9th European Conference on Software Maintenance and Reengineering, CSMR 2005, pp. 256–264 (2005)
7. Cunningham, W.: The WyCash portfolio management system. SIGPLAN OOPS Mess. 4, 29–30 (1992)
8. Fowler, M.: Technical Debt (2009), http://martinfowler.com/bliki/TechnicalDebt.html
9. Sterling, C.: Managing Software Debt: Building for Inevitable Change. Addison-Wesley Professional (2010)
10. Lim, E., Taksande, N., Seaman, C.: A Balancing Act: What Software Practitioners Have to Say about Technical Debt. IEEE Software 29, 22–27 (2012)
11. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 47–52. ACM, New York (2010)
12. McConnell, S.: Technical Debt, http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (2007)
13. Klinger, T., Tarr, P., Wagstrom, P., Williams, C.: An enterprise perspective on technical debt. In: Proceedings of the 2nd Workshop on Managing Technical Debt, pp. 35–38. ACM, New York (2011)
14. Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetro, A.: Using technical debt data in decision making: Potential decision approaches. In: 2012 Third International Workshop on Managing Technical Debt (MTD), pp. 45–48 (2012)
15. Seaman, C., Guo, Y.: Measuring and Monitoring Technical Debt. In: Zelkowitz, M. (ed.) Advances in Computers. Academic Press (2011)
16. Zazworka, N., Seaman, C., Shull, F.: Prioritizing design debt investment opportunities. In: Proceedings of the 2nd Workshop on Managing Technical Debt, pp. 39–42. ACM, New York (2011)
17. Vetrò, A.: Using automatic static analysis to identify technical debt. In: Proceedings of the 2012 International Conference on Software Engineering, pp. 1613–1615. IEEE Press, Piscataway (2012)
18. Fontana, F.A., Ferme, V., Spinelli, S.: Investigating the impact of code smells debt on quality code evaluation. In: 2012 Third International Workshop on Managing Technical Debt (MTD), pp. 15–22 (2012)
19. Guo, Y., Seaman, C.: A portfolio approach to technical debt management. In: Proceedings of the 2nd Workshop on Managing Technical Debt, pp. 31–34. ACM, New York (2011)
20. Gaudin, O.: Evaluate your technical debt with Sonar (2009), http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar/

21. Anderson, D.: Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press (April 7, 2010)
22. Gaillot, E.: What is Coding Dojo (2012), http://codingdojo.org/cgi-bin/wiki.pl?WhatIsCodingDojo
23. Werbach, K.: Gamification, University of Pennsylvania (2012), https://www.coursera.org/course/gamification

Appendix A - Technical Debt Criteria

	Red	Yellow	Green
Automatic Construction	No build tool used	Build tool is used but build is dependent on local configuration	Build tool is used and build is not dependent on local configuration
Continuous Integration	There is no job in the CI Tool	There is a job scheduled in the CI Tool	There is a job scheduled in the CI Tool and the team is committed to keep the build working
Automatic Deployment	Deployment is manual	Deployment is an automated process using a build tool command	Deployment is an automated process using the CI Tool
Continuous Delivery	Release to staging environments is manual	Release is an automated process for validated artifacts using a build tool command	Release is an automated process using a build pipeline
Style Good Practices Bugs	No static analysis tool used	Static analysis tool is configured with static rules	Static analysis tool is configured and the team is committed to keep high levels of rules compliance
Architecture	No architecture analysis tool used	Architecture analysis tool is configured with architectural (dependency) rules	Architecture analysis tool is configured and the team is committed to keep high levels of rules compliance
Tests: Unit/ Integration/ Acceptance/ Performance/ Load/ Security	No Tests	Some Tests	There are tests in a level that the team is comfortable with

Statistics	No statistics on the code quality	Statistic on the code quality are collected	Statistic on the code quality are collected and the team is committed to keep high levels of quality
Monitoring	No monitoring	The monitoring tool is configured to alert the team when the application is not responding	The monitoring tool is configured to alert the team when the application or any of its dependences are not responding