A Systematic Method to Evaluate the Software Engineering Practices for Minimizing Technical Debt

Vinay Krishna
SolutionsIQ India Consulting Services Pvt Ltd
14th Cross, 10th Main, Malleswaram
Bangalore, India

Anirban Basu, PhD
APS College of Engineering
Somanahalli, Kanakpura Road
Bangalore, India

ABSTRACT

Often we find it difficult to incorporate any changes in a software project during later phases of its development, or during post-delivery maintenance. Primary reason for this is inflexibility in design and code which makes it difficult for changes to be incorporated. This inflexibility substantially increases the cost of making changes and this metaphor has been termed as Technical Debt [1].

While Technical Debt cannot be eliminated completely, its burden needs to be reduced. Many practitioners, especially from agile community, have suggested some practices to avoid or eliminate Technical Debt. This paper discusses on a systematic method to evaluate the six software engineering practices that a developer can follow to minimize Technical Debt. These practices have been used and found to be effective when implemented in projects as discussed here.

General Terms

Technical Debt, Code improvement, Refactoring

Keywords

Technical Credit, Living Budget

1. INTRODUCTION

One major technical challenge in most of today's software projects is introduction of unnecessary complexity in design and code, knowingly or unknowingly [1] [2]. System requirements mature with time, business requirements change with market dynamics and evolution of technology requires developing the requirements again. Incorporating desired changes at a late stage of software development require modification to the design and code. As change is inevitable, developers make quick and dirty changes in design and code to meet customer's expectations without any disruption in the schedule. Such unplanned changes done by the developers add complexity to the code. There are also situations when developers unknowingly make the code messy by not abiding to the prescribed coding standards, by incorporating changes in a hurry and by making over commitments without understanding the ramifications. Whether it is inadvertent or deliberate, such changes cause "stiffness" in code and gradually a situation is reached when making further changes in the code become extremely difficult. This state in the code leads to a situation termed as Technical Debt [1]. Many times reason is over use of Technical Credit which involves adding features in design and code to incorporate over-anticipated changes. A balance is required between essential and accidental complexity in design as well as in code and optimizing the design and the code is the biggest challenge facing software architects.

Although, IT community understand the ill effects of Technical Debt little has been done to minimize it. Software engineers, who are the key stakeholders in software development, can play an important role in minimizing it. This paper prescribes robust software engineering practices for reducing Technical Debt. Implementing the practices proposed in this paper require discipline and strict adherence to defined practices. The prescribed practices were implemented in real life software projects and initial results have been presented in [3]. Since then the techniques have been implemented in more projects and data collected shows that the proposed techniques can substantially reduce the Technical Debt.

2. BACKGROUND

According to James Higgs [4], "All projects incur Technical Debt, and that's not a bad thing". He has explained different grades of Technical Debt and how we can overcome it.

Practitioners from the software development community have suggested many good practices to reduce Technical Debt [2] [4] [5] [6]. As described in Table 1 these practices can be classified into 3 groups: Practices to Identify, Practices to Classify and Practices to Reduce.

Table 1. Classification of practices to reduce Technical Debt

Category	Description	Practices
Identification [2][4][6]	Contains practices to identify	Poor code quality Insufficient code coverage Inadequate documentation
Classification [2][4][5][6]	Contains practices to Classify	Knowingly/Unknowingly Short term/Long Term Prudent and Reckless Debt Strategic/Non-strategic 4 grades of debt
Reduction [2][6]	Contains practices to Reduce by	Refactoring Test Driven Development Code reviews/ Audit Pair programming Continuous Integration Best Practices/ Coding Standard Evolutionary design

Practices related to Identification provide the developer ways to identify Technical Debt in the code whereas the practices in the Classification category help in understanding the reason. Reduction practices have been prescribed to minimize the debt identified. However, we find that although the practices suggested to identify, classify and reduce Technical Debt are effective to some extent, adoption of these practices in real life software projects are always a challenge and it requires mindset change at developer level. This paper proposes software engineering practices along with evaluation process which have been found to be more effective in practical situations and help a lot in adoption of above reduction techniques.

3. SOFTWARE ENGINEERING PRACTICES FOR REDUCING TECHNICAL DEBT

Although the benefits of Test Driven Development and other good practices [7] are well established, developers feel that effective methods [8] are still missing to reduce Technical Debt. With experience on working on several projects, we were able to identify six software engineering practices discussed below that can be used to reduce the Technical Debt. These practices are discussed along with situations where it can be applied.

3.1 Practice 1: Determine one's living budget

Description: Minimal output (complete or part of feature implementation in terms of code) that needs to be produced in a day to meet the deadlines is defined here as "living budget". Every developer must know his/her living budget and needs to be introduced in Work Management Plan. The concept of "Living Budget" is as follows:

When one plans his/her development work, one must estimate and plan for code review by self and by refactoring. So if one plans for z hours of work in a day (normally z=8) one should plan to spend some portion say x hours for development and y hours for review and for refactoring the code. The value of x and y should be determined by the developer.

1 day = z hrs

$$1 \text{ day development} = \underbrace{x \text{ hrs development} + y \text{ hrs review and refactoring}}_{\text{Living Budget}}$$

where x hrs + y hrs = z hrs

Recommendation: One should include time for code review and refactoring in work plan. Sprint planning practice of Scrum have been found to be useful as team availability and daily hours available of each team member is known in advance. Besides, we suggest following approaches:

i) Efficiently utilize extra/free time
In some projects we get extra time either due to early completion of assigned tasks or due to some other reasons. In such situations, this time should be used effectively for Technical Debt reduction. This additional time should be in addition to the time budgeted in Practice 1.

ii) Be Self-organizing
One must be able to manage his living budget, and keep track of all time and delivery commitments. We should update code regularly and keep monitoring so that undesirable practices do not recur. Team members should be empowered to do task selection, estimation etc. There are many Scrum practices such as Daily standup and retrospective which help to achieve these.

3.2 Practice 2: Smell one's own code

Description: Code should be reviewed to find out areas where defects are likely to occur and there is possibility of having redundant code. Steps should be taken to reduce/remove unwanted code in these areas, even if it means removing certain portions introduced due to over anticipation. Although this is well understood and easy to do, it is hard to follow. Normally developer finds very less time or no time to review/smell one's own code since he/ she is always struggling to meet the deadlines. Following Practice 1, i.e., "Determining one's living budget", helps to plan for this activity.

Recommendation: For following this practice, first define coding standards and best practices and make the team aware of these. A check list should be created and developer should use it to make sure that defined coding standard and best practices are adhered to. A manual process is hard to follow and it is beneficial to use some static code analysis tool that can be used to find out deviation from standards and best practices. However we still need to apply manual effort to review the code in order to refactor it.

3.3 Practice 3: Make optimal use of Technical Credit

Description: Introducing complexity for anticipated requirement in design and code is termed here as Technical Credit. This adds complexity to design and code which may not be required eventually. This is very important aspect in coding and unless one is sure about future needs, one should not introduce flexibility due to anticipation.

If one introduces additional complexity in the code to cover some un-practical scenarios, it is necessary to deal with these and get to the causes. Introducing unnecessary complexity makes the code more complex and rigid and increases Technical Debt. It is better to remove such additional complexity as early as possible.

Recommendation: We need to encourage all members in the development team to discuss all issues in order to avoid guessing customer requirements and over anticipation. The following approach is recommended:

i) Start Refactoring the Technical Credit portions
The portions of the code having Technical Credit are to be found. Refactoring to improve the code should start after that. More attention should be given to the portions where additional code has been written due to anticipation. These are portions with Technical Credit. We should refactor only one portion at a time until it is improved and look for reduction in Technical Debt. Refactoring on one portion (section) of the code is better than refactoring on several parts of the code at the same time.

ii) Take help from others in design and coding related obstacles
We have found that frequently we spend time on issues which have already been solved by someone else or can be done quickly by a person with the necessary expertise. But we avoid seeking help from them. Pair programming is the best option to avoid such situation. However if we cannot practice pair programming, we need to encourage open communication.

iii) Stop Keeping up with Joneses
Avoid blindly following others' designs, patterns, codes and libraries unless one really needs them. Ask suggestions from all but accept the best one suitable.

3.4 Practice 4: Follow Best Practices and Coding Standards

Description: Use recommended Coding Standards/Guidelines. This is the best way to get code back on track. If one portion of the code e does not adhere to the standards/guidelines, one needs to modify it. Adhering to same coding standard and best practices makes the design and code more readable, maintainable and it's easy to understand other's design and code.

Recommendation: Define the best practices and Coding Standards and share them with the team. Check if any static code analysis tool can be used for review and to quickly find out deviations or shortcuts.

3.5 Practice 5: Increase productivity with Quality in mind

Description: Always focus on quality and not on speed. Never measure productivity in terms of quantity but in terms of quality and importance. If quality of design and code is not taken care always by the team, then the productivity of team starts decreasing. In the lack of adequate level of quality, both design and code gradually becomes messy. Over time mess becomes so big, deep and tall that it's very hard to clean. Eventually productivity reaches to zero [10].

Recommendation: Test driven development is one of best practices to increase the code quality. Maintaining product backlog with proper order is also a good practice to get important items done first. Continuous integration is another good practice, as we make changes in our code apart from unit testing. Always do an integration testing to make sure it did not break others code.

3.6 Practice 6: Learn continuously

Description: Learn techniques continuously and apply it to improve code. Plenty of resources are available to enhance one's knowledge.
Recommendation: Impart proper training to the team on code refactoring and share good resources with them. Encourage continuous learning and experience sharing within the team.

There is always scope for improvement and continuous learning helps. Never give up on learning emerging coding standards, best practices, refactoring techniques etc.

Discuss with the team technical updates, any new special defect or fix that has been encountered or used by anyone and keep the meetings less formal and encourage team member to share his/her experience and learning.

4. APPLICATION ON PROJECTS

Although SQALE method [9] has been proposed, measurement of Technical Debt is not easy.

We have used some questionnaires for analyzing the impact of practices used to reduce the Technical Debt. The responses given to a set of questions are collected before and after applying the six practices. This helped us to find out the effectiveness of the practices in minimizing Technical Debt. We chose three different projects in banking domain for studying the effectiveness of the practices. Below are the questions with respective scores used to assess the project teams. The desirable values of the each question are 1 or 2.

Table 2. List of questions along with score to analyze the impact of six practices used

Q. #	Questions	Score
1	Does anyone review the deliverables? Who reviews it?	+2: yes by peer +1 : yes, by tech lead or senior BA -1 : yes, by expert -2 : yes, by management. Or no control.
2	How does the build managed and integrated?	+2 : Multiple commits on a main branch (or short-lived feature branches) + multiple integrations per day +1 : Daily commit per branch + weekly integration 0 : All sources are under version control, integrated once per release -1 : All applications source files except SQL are under version control , integrated once per release -2 : Some application source files are not under version control , integrated once per release
3	Is there any coding standard or best practice exists for the team	+2 : Everyone uses and edit it and it's enforced with some commit hooks +1 : Everyone uses the standard and applies it 0 : Exists and applied a little -1 : Exists but not applied -2 : No coding standards
4	How does the decision being made within the team	+2 : Constructive disagreement +1 : Consensus -1 : By majority -2 : The team manager makes the decision
5	Who estimates the effort and assign the tasks within the team?	+2: whole team +1: team experts -1: (Project) Manager + team experts -2 : (Project) Manager alone
6	How do you insure that new functionalities can be easily added (technically) to the product, all along the product life?	+2 : All team members refactor continuously +1 : The tech lead refactors continuously 0 : We clean some code monthly -1 : We clean some code a few time per year -2 : We never refactor anything
7	How do you test a feature?	+2 : Automated tests written before the development regarding feature behavior. +1 : Automated tests written before the development regarding technical behavior. 0 : Automated testing regarding the expected behavior. -1 : Manual testing of the expected behavior -2 : Manual testing of the technical impacts

Q. #	**Questions**	**Score**
8	What is the level of non-expected behaviors in test environment?	+2 : Close to zero (We discover everything in development environment because we use BDD or TDD) +1 : Less than 1 defect per feature. Good collaboration between BA, Dev and Biz -1 : More than 1 defect per feature. Regressions are seldom -2 : More than 1 defect per feature. Regressions are not seldom
9	Are your non production environments representative from the production environment? Are deployment procedures the same?	+2 : Mostly identical, non discriminent adjustments (minimal variable adjustments, less amount of nodes in a cluster, less IOs bandwidth...) +1 : Quite representative with few, well known and controlled adjustments (single node compared to server farm or cluster) -1 : Few convergence, major or uncontrolled differences remains -2 : Completely different architectures/technos/procedures
10	How is the knowledge flow inside the team?	+2 : Flawless circulation of the knowledge between team members +1 : Knowledge circulation is quite flawless except for a few domains 0 : Some domains are carried by an expert, for other domains, knowledge circulation is flawless between team members -1 : Each domain is carried by an expert -2 : Only one expert in the team, he answers most (if not all) of the questions
11	What is a bus/truck factor in the team	+2 : Nothing, business as usual with throughput slowness at worse 0 : Some domains can't be maintained anymore -2 : Our clients complains when a senior/key team members not available in the team
12	How transparent am I when I made a mistake?	+2, I feel comfortable with the team enough to be transparent and tell when I made a mistake. The same mistake never occurs twice -2 : I try avoiding talking about my mistakes because of the consequences it may have

Above questions are used to find out the effectiveness of the proposed practices and evaluate them. The same set of questions are used in three different projects and scored are captured before and after applying the proposed practices. Table 3 gives the scores captured before and after adopting the six practices in three different projects. It is evidently established that when team has not adopted the six practices then they scored poorly and when they adopted six practices they scored much better. Eventually with the questions and respective scores, it demonstrates clearly that team was able to minimize the technical debt efficiently by adopting the proposed six practices.

Table 3. Score captured before and after adopting the six practices

Practice	**Q. #**	**Project 1**		**Project 2**		**Project 3**	
		Before	**After**	**Before**	**After**	**Before**	**After**
Practice 1, 3, 6	1	-1	2	1	2	-2	2
Practice 5	2	0	2	-1	2	1	2
Practice 2, 4	3	0	2	0	2	-1	2
Practice 3, 6	4	-2	1	-1	1	0	2
Practice 1	5	-1	2	1	2	-1	2
Practice 3, 5	6	-1	2	0	2	1	2
Practice 5	7	-1	2	-1	2	-1	2
Practice 5	8	-1	2	-2	1	-2	1
Practice 4	9	1	2	-1	2	-1	2
Practice 1, 3, 6	10	0	2	-1	2	-2	1
Practice 1, 3, 6	11	-1	2	-2	1	0	1
Practice 3, 6	12	-1	2	-2	2	-2	2

Besides minimizing Technical Debt, we also observed significant improvement in number of defects found during UAT after applying these practices effectively as shown below in Figure 1.

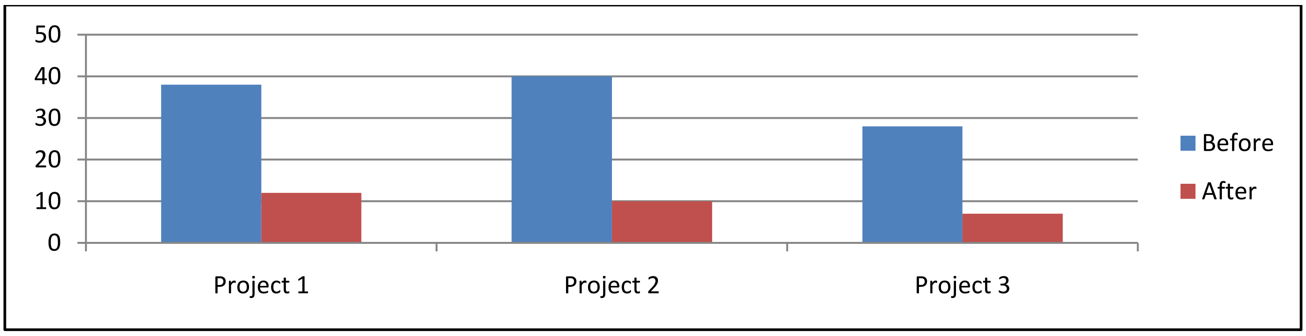

Figure 1: No. of defects found before and after applying the six practices

5. CONCLUSION

While it is difficult to eliminate Technical Debt completely, it is abundantly clear that large amount of debt can lead to failure or substantial loss in terms of extra effort and rework needed to make changes to meet customer expectations. As a developer we should minimize Technical Debt as much as possible. This paper suggests software engineering practices to reduce Technical Debt. The practices have been found to be effective based on the authors' practical experience on application on real life projects. The six software engineering practices proposed in this paper are being applied on more projects of different categories and sizes to check their robustness.

6. ACKNOWLEDGMENTS

Our thanks to the teams and experts who have contributed towards development of the template.

7. REFERENCES

[1] W. Cunningham, The WyCash Portfolio Management System, OOPSLA, 1992;http://c2.com/doc/oopsla92.html

[2] S. McConnell, Technical Debt, 2007; http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[3] V. Krishna and A. Basu, "Minimizing Technical Debt: Developer's Viewpoint", in Proc ICSEMA 2012, Chennai, Dec 2012

[4] J. Higgs, The Four Grades of Technical Debt, 2011; http://madebymany.com/blog/the-four-grades-of-technical-debt

[5] M. Fowler, Technical Debt Quadrant, 2009; http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html

[6] T. Theodoropoulos, Technical Debt Part1-4, 2012; http://blog.acrowire.com/technical-debt/technical-debt-part-1-definition

[7] V. Krishna, My Experiments with TDD, ScrumAlliance, 2010; http://www.scrumalliance.org/articles/357-my-experiments-with-tdd

[8] D. Laribee, Using Agile Techniques to Pay Back Technical Debt, MSDN Magazine, December 2009; http://msdn.microsoft.com/en-us/magazine/ee819135.aspx

[9] J. Letouzey, The SQALE Method, January 2012; http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf

[10] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Aug 2008, Prentice Hill, page 4