HOW TO MINIMIZE TECHNICAL DEBT?

By Madhura Oak
Project Manager,
Intellect Design Arena Ltd.
1. Abstract
“Clean code always looks like it was written by someone who cares.” – Michael Feathers, author of Working Effectively with Legacy Code.

Gartner has estimated Global ‘IT Debt’ to grow by $1 trillion in 2015.

2. Introduction
Ward Cunningham, inventor of Wiki and pioneer in extreme programming and design patterns, introduced very a well known metaphor - Technical Debt which refers to the consequences of poor software design, architecture or development. The pace with which IT Industry has boomed in the last 15 years has resulted in creating technical debt worth millions of dollars. The lack of standard practices, design patterns and mature processes could have created it during the initial years but the race to deliver software applications in a quick and dirty way without keeping a perspective on their scalability and maintainability, is a major contributor to accumulation of so much technical debt. Like financial debt, technical debt incurs cost due to extra effort needed to maintain bad code and with time this debt goes on rising exponentially.

Technical debt has two impacts – it increases costs of maintenance projects and can also cause bad reputation and loss of business.

If an important client asks you to deliver a customized feature in your software product so that he could gain a competitive advantage and you discover that it could not be delivered within the specified time frame due to technical debt in your product, you run the risk of losing a client. It is said that losing one client leads to losing five clients. It takes years to create a good reputation and consciously build a brand image for any organization. You would certainly not like to risk it due to your technical debt.

Technical debt can be categorized into code debt, design debt, technology debt and IT infrastructure debt.

This paper focuses on code and design debt. It describes how technical debt increases project costs. It describes the causes of technical debt and suggests actions that should be taken by various project stakeholders to minimize technical debt.

3. How Technical Debt increases project costs?

1. Decreases code readability
It is difficult to read code with anemic domain models, absence of code modularization and meaningful names for classes, methods or variables. At times, the code is implemented in Object Oriented Programming (OOPs) technologies but written with procedural style of programming.

2. Increases application maintenance effort
Code readability impacts its maintenance directly. It takes longer to modify code that is difficult to read. Increase in maintenance effort increases project costs.

3. Increases project schedules
Increased application maintenance effort may require additional developers or increase in project schedules.

4. Increases time to market
Increased project schedules increases time to market of a product. This could be a risk in gaining competitive advantage.

5. Decreases productivity
Productivity of developers working on code that is difficult to read diminishes as the project scope grows over a period of time (Figure 1).

Figure 1: Productivity of developers working on project with Technical Debt
6. Increases employee attrition
“Technical Debt can kill productivity, making maintenance annoying, difficult or in some cases impossible. Beyond the obvious economic downside, there’s a real psychological cost to technical debt. No developer enjoys sitting down to his computer in the morning knowing he’s about to face impossibly brittle, complicated source code. The frustration and helplessness thus engendered is often a root cause of more systemic problems such as developer turnover – just one of the real economic costs of technical debt.” – David Laribee.

7. Increases probability of defect induction
Code which is difficult to read is more error prone. If you are not using Test Driven Development (TDD), then the probability of defect induction is higher in code that is difficult to read and it is directly proportional to the project scope and complexity.

8. Increases risk of application turning into a legacy application
Legacy applications are the ones which cannot be modified but are still in use. Decreased code readability or use of obsolete technologies (technology debt), increases the risk of your application turning into a legacy application.

9. Decreases Return on Investment (ROI) of code
Decreased code readability and increased maintenance effort decreases the ROI of code.

4. Causes of Technical Debt

1. No Code Reviews
Project teams which do not conduct code reviews are at the highest risk of accumulating technical debt.

2. Lack of experience in project domain / framework / technology
Developers who have less experience in project domain or framework or technologies can potentially increase the technical debt than the experienced ones.

3. No knowledge of Clean Code
Majority of the IT workforce today are programmers at the position of Software Engineer, Senior Software Engineer or Team Leader with less than ten years experience. Many freshers who join IT organizations are unaware of Clean Code as it is still not part of the curriculum in many educational institutions. There is a huge difference in writing code and writing production quality code. Software organizations often conduct trainings on programming languages for the fresh recruits but very few train them on Clean Code. Lack of understanding of Clean Code increases technical debt.

4. Lack of Learning
Very few software professionals at middle management levels take initiative in learning unless it is mandated by their employers. It is often observed that workforce at middle management levels stagnate if their employers do not stress on mandatory learning and certifications. Since this category of workforce also acts as an influencer in making technology decisions for a project, lack of knowledge of new technologies in marketplace and updates to a technology could lead to technology debt.

5. Metaphor of broken windows
Many programmers tend to follow the existing coding style in a software project. Thus, the mess in an already messy software application keeps growing.

6. Squeezing Estimation of Development Tasks
Though people realize the importance of code reviews, very few project teams follow them rigorously. Code review is considered as a part of development activity and is usually done towards its end. Squeezing estimation for development usually eats up time allocated for code review process. Estimation done without considering variance according to the cone of uncertainty (Figure 3) and time spent on unproductive tasks are also responsible for steep development deadlines.

The cone of uncertainty depicts the variance that can occur in estimation during the various project phases. For example, a requirement estimated to be developed in 10 weeks during the initial product definition phase could take anywhere between 60% to 160% of the estimated time i.e. 6 weeks to 16 weeks.
Following are some of the unproductive tasks on which the development time is spent:

a. Frequent setup of development workspace and testing environments.

b. Resolving code conflicts and compilation errors after receiving code updates from configuration management system.

c. Configuration for testing which includes creation of test data for every testing activity.

7. No / Insufficient documentation and knowledge transfer

Project documents such as Product Specifications, Architecture Design Document (ADD), Functional Specifications and Application Developer Guide help in understanding the application domain, architecture and development practices followed within the application. Lack of these documents or outdated documentation and insufficient or no knowledge transfer leads to the developers following their own way of coding which adds to the technical debt.

8. Insufficient Testing

The business logic needs to be thoroughly tested for all its use cases every time it is modified. This testing includes both positive as well as negative test scenarios. Regression testing requires enormous manual testing effort if test automation is not used. Since it is practically impossible to allocate budgets for rigorous manual testing effort, there is possibility of gaps in testing.

9. More emphasis on costs and schedule by senior management than code quality

During project planning, costs and schedule are given more emphasis than code quality. Often, quality is an outcome but not the main focus during project execution. Code quality is often compromised as you decrease costs and schedule. Some project managers, particularly the ones who have never played technical roles, fail to understand that once the code quality is compromised, maintaining the code in the long run becomes a nightmare.

5. How to minimize Technical Debt?

The actions that can minimize Technical Debt are described below:

1. Development

1.1. First time right code

Ensure that code quality is right the first time it is written as technical debt keeps accruing with time. Many software organizations have adopted the first time right culture.

1.2. Use Clean Code, design patterns and standard coding practices

Use of clean code, design patterns and standard coding practices increases code readability and makes application maintenance easy.

1.3. Test Driven Development (TDD)

In TDD, test cases are present for all test scenarios and they are executed after every enhancement to business logic. Functional testing can be automated using TDD. TDD significantly reduces the cost of manual regression testing and the risk of unidentified defects in code.

1.4. Maintain a library of reusable code within organization / business unit

A library of tested reusable code helps in reducing the development and testing effort. Reusable code can be provided as off-the-shelf pluggable components or web services.

2. Testing

2.1. 100% test coverage

The test scenarios should cover 100% of the application functionality. They should cover integration testing as well as the user interfaces. Reviews should be conducted to assess test coverage. Testing should include both positive and negative test scenarios.

2.2. Automated testing and Continuous Integration

Automated testing can be used to test user interfaces of an application. Regression testing done using automated testing brings tremendous cost savings in maintenance projects as compared to manual testing.
Continuous Integration (CI) in combination with running automated test cases during every build ensures that the existing business logic works fine. Since builds are built multiple times in a day in CI, a build failure immediately highlights a defect. Appoint a Build Manager who is in charge of monitoring builds and responsible for getting the build conflicts resolved from the team.

Many organizations are implementing DevOps in which the development and testing environment is similar to the production environment.

2.3. Reporting and tracking of defects
All technical defects should be promptly reported and tracked using a defect tracking tool. The defects can be of three types – defects that occur during setup of application, defects that occur after the application is deployed and defects in the application found during development or testing. Steps to resolve frequently occurring defects should be documented.

3. Processes

3.1. Make Code review an independent and mandatory activity
Code review should be made an independent activity. It should not be an implicit and concluding part of development. Explicit time estimation should be done for code reviews and they need to be done diligently. In a few organizations, audits are conducted by Software Quality Assurance (SQA) teams to assess the code quality. These audits should be made mandatory for the SEI CMMI Level 5 level companies. Adherence to coding standards should be evaluated in these assessments.

Similarly, design reviews for low level designs and high level design should also be made mandatory.

3.2. Documentation
Project documents such as project specification, use case document, developer guide and Architecture Design Document (ADD) aid in understanding the application. Like code reviews and audits, documentation should also be reviewed and audited to ensure that the relevant project documents are updated for every enhancement.

3.3. Make learning mandatory
Many employers mandate 80 hours of learning per year for an employee. Project managers/team leaders need to ensure that their developers are trained on Clean Code, coding standards and the technologies being used in the project. A good way to validate learning is technical certifications.

3.4. Identify technical debt and plan to reduce it
There is a high probability of technical debt in maintenance projects as awareness about best coding practices such as design patterns and Clean Code did not exist ten years ago.

Acknowledge presence of technical debt and use metrics to identify areas of code in which it exists. Plan to reduce technical debt in the areas with highest impact such as domain model or business logic layer.

Approving budgets from senior management to reduce technical debt is a challenge for a project manager. Project managers can convince the project stakeholders by projecting costs of application maintenance with technical debt in comparison to maintenance costs after reducing technical debt. Savings in effort leads to reduced costs and faster time to market. It becomes easy to convince senior management by projecting savings.

As getting rid of technical debt does not bring any direct business benefit apart from future cost savings, it may not be possible to do it in one go due to budget constraints. Using an iterative plan to get rid of code debt is preferred. Technology debt may or may not be minimized at once.

3.5. Effective communication with project stakeholders
Most of the software companies use a tool for tracking requirements and defects. Such tools also act as a communication channel between various project stakeholders. Project managers need to ensure that information about change in application architecture, database change log, changes in development, integration and testing environments is communicated across the entire team. There are less communication gaps in project teams which use push communication instead of pull communication. Effective communication minimizes assumptions, thus minimizing the chances of technical debt.
3.6. Identify and minimize unproductive tasks
Unproductive events such as installation and setup of project workspaces, resolution of build errors, frequently occurring application errors, network or database downtime etc. should be tracked for every resource in the project. The purpose of this metrics is to identify collective team effort wasted in unproductive work. Frequently occurring unproductive events can be minimized.

4. Tools

4.1. Use tools to identify technical debt
Tools such as SonarQube or pmd help in identification of code debt.

4.2. Use software versioning and revision control tool
Software versioning tool helps to keep track of simultaneous code changes by multiple developers. It ensures that pieces of code are not overwritten and lost during a code update. It helps to keep track of updates and maintain codebases of different software versions effectively.

4.3. Development and Defect tracking tools
Use of development and defect tracking tools give better visibility of tasks and defects assigned to developers and makes their tracking and closure easy. Apart from maintaining transparency in the organization, they also ensure that tasks and defects are not missed.

5. Design

5.1. Use Layered Application Architecture
Maintaining applications with layered architecture is easy since it promotes Separation of Concerns (SoC). Each layer has a unique responsibility and it only interacts with the adjacent layer. A Java EE application normally uses the following three layers – Presentation, Business and Integration.

5.2. Use strong Domain Modeling and Domain Driven Design
If you are using OOPs, then ensure that entities in your software design are distinct i.e. have independent identity and are related to each other. The entities should not be anemic. The functions of an entity as well as their relationships with other entities should be included within the entity class. Following are the advantages of using strong domain modeling:

a. Ease of maintenance - The code is easy to maintain since all the code related to an entity exists within the entity class.

b. Strong domain models make application maintenance easier and thus less costly.

5.3. Plan and Design by consensus
Project managers or architects are usually responsible for project planning and application design. Though high level estimation or design decisions could be taken by an individual, it is always better to have a consensus of project stakeholders. Toyota’s management principle of making decisions slowly by consensus, thoroughly considering all options and implementing decisions rapidly has been proved very effective in manufacturing companies and it could be applied in software organizations as well.

Group estimation approaches such as Wideband Delphi, planning poker and story points are considered as most efficient estimation techniques.

6. Technology

6.1. Maximize use of well-known and commonly used open source frameworks
Projects which use well-known open source technologies and frameworks such as JPA, Struts, Spring etc. are easy to maintain since availability of developers trained in them is more in the market. Trained resources are more productive than the ones who are not trained or with insufficient training.

In case the project stakeholders decide not to use a popular framework or to use their own framework, then the developers working on maintaining such projects should be trained on the project specific framework.

Figure 4 further categorizes these actions into the ones to be taken by various project stakeholders.
### How Project Stakeholders can reduce “Technical Debt”?

<table>
<thead>
<tr>
<th>Senior Management</th>
<th>Project Manager / Team Leader</th>
<th>Architect / Technical Leader</th>
<th>Application Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processes</strong></td>
<td><strong>Processes</strong></td>
<td><strong>Design &amp; Development</strong></td>
<td><strong>Processes</strong></td>
</tr>
<tr>
<td>✓ Make Code Audits mandatory</td>
<td>✓ Code Reviews</td>
<td>✓ Layered Application Architecture</td>
<td>✓ Clean Code</td>
</tr>
<tr>
<td>✓ Make Documentation mandatory</td>
<td>✓ Product/Project Documentation</td>
<td>✓ Strong Domain Modeling and Domain Driven Design</td>
<td>✓ First time right code</td>
</tr>
<tr>
<td>✓ Emphasise on Test Automation</td>
<td>✓ Identify and Plan to reduce technical gaps of team</td>
<td>✓ Design Patterns and standard coding practices</td>
<td>✓ Use Design patterns</td>
</tr>
<tr>
<td>✓ Bring Learning culture</td>
<td>✓ Document and Plan to reduce Technical Debt</td>
<td>✓ Use open source frameworks</td>
<td>✓ Use standard coding practices</td>
</tr>
<tr>
<td>✓ Promote Reusable Code Repository within and across Business Units</td>
<td>✓ Ensure effective communication between project stakeholders</td>
<td>✓ Use reusable code</td>
<td>✓ Report Defects</td>
</tr>
<tr>
<td>✓ Promote use of open source frameworks</td>
<td>✓ Identify and minimize unproductive tasks</td>
<td>✓ 100% coverage in unit testing</td>
<td>✓ ReportTechnical Debt</td>
</tr>
<tr>
<td>✓ Promote use of design patterns and standard coding practices</td>
<td>✓ Minimize Technology Debt</td>
<td>✓ Learning</td>
<td>✓ Task and Defect tracker</td>
</tr>
<tr>
<td>✓ Minimize Technology Debt</td>
<td>✓ Promote use of open source frameworks</td>
<td>✓ Learn Technology Trends, new technology innovations, changes in regulations and compliances</td>
<td>✓ Tools</td>
</tr>
<tr>
<td>✓ Develop use of Task and Defect tracking, continuous builds and Documentation</td>
<td>✓ Promote use of design patterns and standard coding practices</td>
<td>✓ Unit Testing</td>
<td>✓ UnitTesting</td>
</tr>
<tr>
<td>✓ Promote tools for effective communication between project stakeholders</td>
<td>✓ Minimize Technology Debt</td>
<td>✓ Automated Testing and Continuous Integration</td>
<td>✓ Automated Testing and Continuous Integration</td>
</tr>
<tr>
<td>✓ Promote use of software versioning and revision control tools</td>
<td>✓ Use Task management and Defect tracking tools</td>
<td>✓ Software versioning and revision control</td>
<td>✓ Software versioning and revision control</td>
</tr>
<tr>
<td>✓ Tools</td>
<td>✓ Use software versioning and revision control tools</td>
<td>✓ Task and Defect tracker</td>
<td>✓ Task and Defect tracker</td>
</tr>
</tbody>
</table>

*Figure 4: Actions for Project Stakeholders to reduce Technical Debt*

### About the author

Madhura Oak works as a Project Manager at Intellect Design Arena Ltd. She has worked on various roles such as Technical Architect, Technical Project Manager, Technical Leader and Application Developer for more than 7 years. She has 13 years experience of working on Java EE and XML technologies. She also holds a Bachelor of Computer Engineering degree from University of Mumbai.

### References

1. Clean code – Robert C. Martin
4. Technical Debt – Martin Fowler
5. Adopt Right Software to Reduce Technology Debt – Madhura Oak
6. Anemic Domain Model – Martin Fowler
8. The Cone of Uncertainty
9. How can startups move fast while minimizing technical debt? William Pietri
10. 9 Quick tips to reducing Technical Debt – Vijay Narayanan
11. DevOps
12. OOP 2014 – Workflows of Refactoring - Martin Flower
15. The Toyota Way – Jeffrey K. Liker

For further information on iGTB please contact Phil Cantor, CMO and Head of Proposals on phil.cantor@intellectdesign.com, +44 7527 396811
THE WORLD'S FIRST COMPLETE GLOBAL TRANSACTION BANKING PLATFORM