Pragmatic Approach for Managing Technical Debt in Legacy Software Project

Rajeev Kumar Gupta, Prabhulinga Manikreddy, Sandesh Naik, Arya K
Corporate Technologies and Development and Digital Platforms
Siemens Technology and Services Private Limited
Bangalore, India
{rajeevkumar.gupta, Prabhulinga, sandesh.naik, arya.kalathiparambil }@siemens.com

ABSTRACT

Tackling the issues of technical debt in a large system in parallel with continuing to enable it to evolve is a challenging problem.

In this paper, we are describing a case study of managing technical debt on a legacy project referred here as Global Configurator Project (GCP) using pragmatic approach. The paper presents holistic lifecycle approach with four stages and various practices in each stage for managing technical debt. Given life cycle approach and practices will be useful for any software project. In particular, these practices will be significant to any legacy project towards repaying debt. These methods can also be applied to continuously improve code quality and product quality. This paper also focus on technical debt user stories to gain business buy-in and share few 'best in market' tools that we used in repaying technical debt. It also focuses on sensitizing developers to the concept of debt and improving their skills. This paper describes the process used by a separate team formed to reduce technical debt in a large legacy system.

The paper targets to the Project Managers, Test Managers architects and Scrum Masters in agile software development.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.8 [**Software Engineering**]: Metrics; K.6.3 [**Software Management**]: Software development, Software maintenance.

General Terms

Management, Measurement

Keywords

Pragmatic, Technical Debt, Lifecycle Approach, Static Analysis, Code Quality, Product Quality.

1. INTRODUCTION

The GCP project developed a product by using a traditional waterfall approach. We released the product in 2008. At present, almost 5,000 end users in the U.S. are using the product. The GCP project team adapted Agile-Scrum in 2012 with an objective to have on time and stable delivery to end users.

Background of GCP:

Project Organization: Development happens majorly in India and some part of it in Germany. Every development team consists of seven to eight members who are architects, developers, and testers. The U.S. team consists of the product manager and the domain experts and they are the interface between the development team and end users.

Technologies: Initially, GCP was developed with client and server based technologies, however, since the last couple of years, GCP is targeting mobile platform with new technologies. GCP is a complex project with few million lines of code has been added using different development frameworks and languages.

Technical Debt: With the intention to increase business, the product manager had taken a strategic call to scale up users. In the year 2013, when users were scaled to nearly 500 from 100, users started facing non-functional issues. Also, new stories were taking more time to market. Our analysis revealed that we have incurred huge technical debt [5] due to various reasons, including faster delivery to market, ignorance, 'code now, clean later', etc. That's when we started focusing on technical debt in GCP project.

2. STRUCTURE OF THE PAPER

The first author has been a part of GCP since 2005 and is presently working as a Project Manager. The second author is working in GCP since 2010 and currently working as scrum master. Other co-authors are part of development teams. They are critical contributors in repaying the technical debt in the project. All the authors have been involved in the managing technical debt. This case study is, therefore, based on all author experience of managing technical debt.

In the case study, we outline the four stages of pragmatic approach for managing technical debt. Each stage discusses steps, levels and practices to repay as well as prevent the debt. To demonstrate our approach, we used a couple of real examples of technical debt in our project and results after its resolution.

This paper does not cover how to solve each type of bad smells; rather focus on explaining a strategic approach for managing technical debt.

3. ABOUT TECHNICAL DEBT

Technical debt is a metaphor coined by Ward Cunningham that helped to think of quick, dirty and messy work done [5]. Technical debt is like financial debt, where interests are paid in terms of extra effort spent in resolving code, design, and architectural smells For example, if a feature can be developed in X time with clean code where as team spend 'Y' effort, then Y-X

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISEC '16, February 18-20, 2016, Goa, India
© 2016 ACM. ISBN 978-1-4503-4018-2/16/02…$15.00
DOI: http://dx.doi.org/10.1145/2856636.2856655

is the extra effort. As this difference increases, it start slowing down future development as well as can potentially de-stabilize production system. Martin Fowler used four quadrants to explains about technical debt [refer Figure 1][14]. He explains that all technical debt is not bad, and there are situations where development team needs to go with fast paced development that incurred technical debt unintentionally.

Over the years, practitioners and researchers have raised questions on which types of smells should be considered as technical debt, like Robert Martin says in Uncle's bob post that mess is not a debt [5]. However, considering that technical debt as just a metaphor, we have treated everything as technical debt where we had spent 'extra' effort for cleaning legacy code, design or architecture.

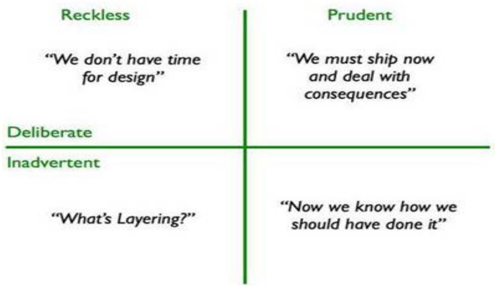
Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 1: Technical debt grid quadrant [20]

Everyday Indicators for Technical debt:

Like other practitioners, we also observed daily behavioral indicatives results in incurring technical debt [33]. Few of indicators are:

i. The only one who can change this code is 'XYZ'. 'XYZ' can be any expert name.
ii. It's ok for now, but we'll refactor it later!
iii. ToDo/FixMe: this should be fixed before release
iv. Let's just copy and paste this part.
v. Let's finish the testing in the next release

Recently, Zengyang Li et. al. has published very comprehensive study on managing technical debt [34] and we recommend it for readers.

4. PRAGMATIC APPROACH

Since we had a legacy code base, our challenges were to adopt an approach that not only repay incurred technical debt but also continuously monitors new technical debts and repay them.

We adapted four stage pragmatic approaches from Tusar et.al. [10] (Figure 2) for repaying technical debt. These stages are explained below:

4.1 Identify

Identify is the first of four stages and is dedicated to identifying technical debt in the project. We used three steps to determine technical debts.

4.1.1 Step1- Tool Identification

We had used various tools available in the market and finalized few tools that were efficiently finding errors related to our field defects and thus used in identifying technical debt in our technologies. We had two types of tools: Runtime analyzer and static code analyzers. Some of these tools are GDI handle [32], IBM Rational Purify[?], Coverity[8], Findbugs[15], Simian[26], source monitor[29], Perfmon[24] and Apache JMeter [4]. We also tried other tools like PCLint, FxCop, Intel thread checker, and Cppcheck[9]. However, we selected only those tools that were able to detect errors related to field issues.

Table 1 shows technical debt topics and associated tool that were relevant to our business needs and are used.

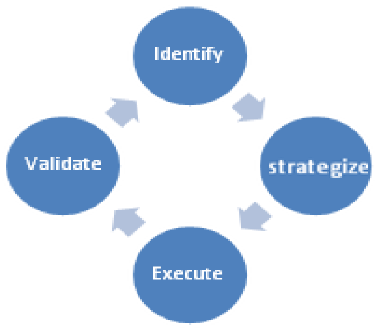

Figure 2: Four stages of pragmatic approach

Table 1: Tools and technical debt topics

Tools	**Types of Errors**
IBM Rational Purify	Memory leak
GDI View	Resource leak
GDI Handle	GDI Leak
Perfmon	Gradual Virtual and Private memory
Coverage Tools (Rational Purify, ECL Emma [13], NCover [22])	Automation Test Debt
Application logs	Redundant server calls
Coverity Errors	Buffer overflows
	Deadlocks
	Error handling issues
	Memory – corruptions
	Memory – illegal accesses
	Null pointer dereferences
	Performance inefficiencies
	Program hangs
	Race conditions
	Resource leak
	SQL Injection
Findbugs-Errors	Malicious code vulnerability
	Multithreaded correctness
	Performance
FxCop-Errors	Efficiency
	Reliability
Coding Best Practices	Design and Architectural Smells

4.1.2 Step2- Tools Execution

We executed these tools for static and runtime analysis of our code and design. These tools give us nearly the complete list of technical debts. Table 2 shows approximate findings from static analyzer [SA: Static Analyzer] tools. 3rd generation static analysis tools like Coverity where we observed that 'False Positive' errors are less than 10% helped us to find prioritized debt with less manual effort. For other tools, we focused on only selective errors based on project needs.

Table 2: Prioritized technical debt from static analyzer tools

	Findbugs	**Coverity**	**FxCoP**
Total SA Errors	234	8000	2600
Prioritized debt Errors	123	1107	800

4.1.3 Step3- Issue Mapping

Development team mapped technical debt with field and quality issues and tagged them as 'technical debt-type' issues, where 'type' was like Memory, Performance, Crash, etc., in our issue management system. As part of our strategy, we focused on most recent issues first as well as keeping the focus on the continuous effort to identify legacy issues related to technical debt. Thus, we resolved most recently pain areas immediately as well as determining backlog technical debt. We analyzed past three years of the data. We had nearly 15% issues (of reported in quality and field) that development team mapped with technical debts as shown in Table 3. These were the actual field and quality defects reported. We had dedicated one sprint for mapping issues to technical debt. Table 2 and Table 3 together gave the total debt.

Table 3: Technical debt mapped with Quality and Field issues

	Memory	**Performance**	**Crash**	**Other Technical Debt**	**Total Technical Debt**
Defects	70	165	483	48	766

4.1.4 Step4- Prioritization

Development team and product owner discussed mapped technical debts from step 2 and step 3 and other technical debt that may cause issues in future. We used different techniques to prioritize debts for including in product backlog as internal stories [Sec 4.3] e.g. technical debt in critical workflows are taken up on higher priority.

4.2 Strategize

The Strategize is the second of four stages and is dedicated to defining strategy related to technical debt in the project.

Scrum Master and Project Manager established two high-level strategies [17] for repaying technical debt and preventing technical debt. Key points of these approaches are mentioned below:

4.2.1 Repaying Technical Debt

We had the following strategy for repaying existing technical debt for the legacy code base. It's like repaying interest effectively for already incurred financial debt.

❖ *Continuous Identification, Prioritization, and resolving identified Technical debt.*

Finding technical debt and solving once will not be enough, unless there is no further development work in current or future. Like in financial debt, it is not enough to pay the interest only once.

❖ *Visualizing debt with Information radiators:*

Technical debt incurred due to various reasons as shown in Figure 1. We required documenting our debt reasons like incorrect coding practices, design smells, inadequate test automation, etc. and visualizing them with the use of information radiators. An Information radiator is a display posted in a place where people can see it as they work or walk by [3]. Information radiators helped us to retain our focus on smells and resolve them effectively.

❖ Continuous Collaboration with Product Owner

Involving and collaborating with product owners was the key factor in repaying technical debt. The development team worked with product owner that helped in prioritization and ranking of identified smells in Identify stage.

❖ *Internal Debt Stories:*

Like functional user stories, we planned internal debt stories for each of prioritized technical debt topics. Internal debt stories were important for buy-in from product owner and business. These debt stories helped developer and business to have a common understanding of technical debt.

❖ *Common Product Backlog:*

It was important to use common product backlog that includes both internal debt stories and functional user stories. That helped in achieving transparency among stakeholders, planning and tracking internal debt stories.

4.2.2 Preventing Technical Debt

Along with repaying existing technical debt, it was more important to apply learning while resolving debt and start avoiding technical debt. Practically, it is not possible to prevent all debt, we planned to repay at the earliest i.e. within the same sprint where it incurred.

We had the following strategy for new development so that technical debt incurred is paid at the earliest.

❖ *Following Boy Scout rule [25].*

This practice ensured that when existing classes or methods are modified, then existing bad smells (like static analyzer issues, code complexity, code clone, dead code, etc.) in those classes or methods are resolved. That means that every time an existing class is modified, it is cleaner than ever.

❖ *Continuous Improvement:*

As per our observations, improving technical skills of scrum team was one of important way to prevent technical debt. We identified technical training based on project need for Scrum teams.

❖ *Technical debt Awareness:*

We believed that technical debt awareness is best among the methods of debt prevention. We planned regular knowledge sharing with development teams of other scrum teams. We shared our findings and corrections on technical debt.

❖ *Improved 'Definition of Done' (DoD):*

We strengthened common DoD for all scrum teams. DoD contains checklist for every story like any code check committed into repository must ensure followings:

1. No open static analysis errors
2. No Open memory leak
3. No degradation in performance compared to baseline
4. Reviewed by experts
5. No open actionable static analyzer violations
6. No Open memory leak violations

❖ *Continuous Improvement in functional test automation*

Functional Testing automation plays a vital role in repaying technical debt for the legacy project. It needs to improve continuously such that it covers important user workflow.

4.3 Execute

Execution is the third stage of four stages and is dedicated to executing action plan defined in strategy [Sec. 4.2].

It is typically very subjective to measure technical debt effectiveness unless it is not associated with business goals. Based on the impact on business objectives, it can be represented qualitatively like Red/Green/Yellow or thumbs up or down [30]. We used a four step execution. These steps are explained below:

4.3.1 Dedicated Technical Debt Scrum Team

We formed a dedicated Technical Debt Scrum Team and named it as TD Team [3]. TD team was like a usual scrum team i.e. cross-functional, multi-skilled and self-managed team, however working on ONLY technical debt stories. The focused business goal for this team was to resolve all those technical debt that are required for scaling up users to around 4000 to more than 5000 users. TD team has six development team members including architects, functional experts, and testers to automate test case and scrum master. We also had a dedicated product owner for TD team. TD team followed three weeks sprint, same as other scrum teams so as the release can be synchronized.

TD Team was a short-lived team and was dissolved after prioritized backlog debt stories were resolved. TD team members joined other Scrum teams as experts on managing debts. They are guiding their scrum team on preventing as well as repaying debts.

4.3.2 Technical Debt Categorization

It was important to categorize identified technical debt so that they can be prioritized and ranked. We used two levels of technical debt categorization.

4.3.2.1 Level 1: Urgent vs. Important

Development team and product owner collaboratively placed all identified technical debts [Sec. 4.1] into Important vs. Urgent metrics [21]. Urgent indicates how soon the fixes are required for users, whereas important shows impact on business-critical workflows. Level 1 helped us to prioritize technical debt using MoSCoW [12] technique.

One of our first few Urgent Vs Important metrics was looking as shown in Figure 3. Smells in each quadrant change are based on discussion with the product owner.

Usually debt that were mapped with user and quality issues were placed in quadrant 4 (MUST) category, whereas debt which were effecting development estimation, were placed in quadrant 2 (SHOULD). Once we resolved most of the user and quality debt issues, the selected smells from quadrant 2 and 3 were set in 4. This prioritization went on time to time.

4.3.2.2 Level 2: Business Value vs. Effort

The Categorization technique discussed in Level 1 has some serious issue like all resource leak were not urgent and important. Also, all identified Coverity and Findbugs errors were not important to resolve in one go. There was some business workflows that were less used by the user compared to others. We used Level 2 categorization to address this issue.

Product Owner and development team placed MUST and SHOULD Technical debts from level 1 categorization into Business value vs. Effort metrics [2]. Business value is calculated in terms of impact on business-critical functional workflows.

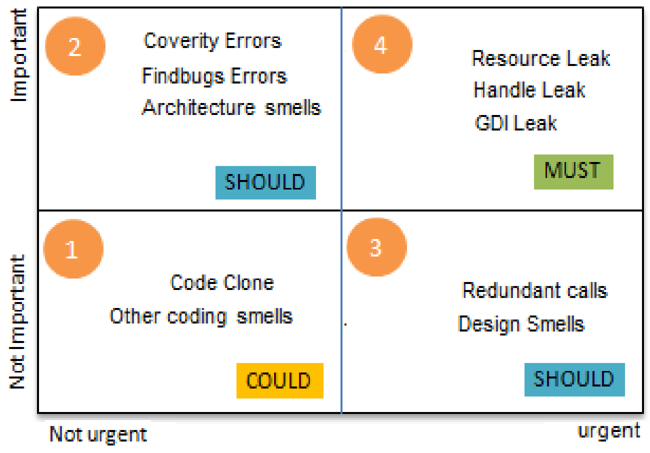

Figure 3: Technical debt in urgent vs. important metrics

Higher the number of impacted workflows, higher the business value of that technical debt. The effort indicates development estimation of resolving the technical debt. Development team first worked on higher business value and lower effort errors.

Level 2 categorization helped in ranking technical debts. The development team worked on high ranked internal debt stories similar to functional stories.

4.3.3 Technical Debt User Stories

Once Technical debts are mapped to workflows, development team created internal user stories [2] for every technical debt topics with estimation. These user stories helped product owner get buy-in from product manager and other stakeholders. This helped us to picture accumulated debt [6]. User stories also contributed to writing test cases to validate and automate.

Few examples are mentioned below:

Example 1: As a development team, I want to resolve 'X' technical debt so that 'A' number of users should able to perform 'Y' set of the workflow for 'Z' time.

Example 2: As a development team, I want to resolve 'X' technical debt, so that 'A' number of users can use the product without the system getting hanged or crashed in any given day.

Example 3: As a development team, I want to resolve 'X' technical debt so that product downtime can be improved 'Z' % times.

Example 4: As a development team, I want to resolve 'X' technical debt so that effort spent on feature development in ABC workflow can be improved by 'Z' %.

4.3.4 Documenting and Visualizing Technical Debt
We have used issue management tool for technical debt management as well. All technical debt user stories are maintained, prioritized and ranked. Additionally, we also used A1 size chart paper and pasted on scrum board to demonstrate technical debt progress. It brought complete transparency and was extremely useful in collaborating for co-located team. Information radiator helped in increasing awareness and motivating other scrum teams on technical debt. The electronic tool helped product manager in the US to visualize technical debt progress.

4.3.5 Continuous Technical Debt Improvement
As mentioned in strategy, best way to repay technical debt is to repay at the earliest. We practiced slack [31] for repaying the technical debt at the earliest. We used slack in two ways, explained below:

4.3.5.1 Slack in every Sprint:
We have four other scrum teams. The primary focus of these teams was on feature development. We called them as the feature team, i.e., Feature Team 1, Team 2, Team 3, and Team 4. Every scrum team has their Product Owner and Scrum Master. Product Owners and Scrum Masters agree to reserve 15-20% team capacity in every sprint for slack. Feature Teams used slack for resolving the technical debt. Usually, Feature team used to pick up high business and less effort debt stories or follow boy's scout rule.

Slack is also used for improving technical skills and knowledge sharing.

4.3.5.2 Slack Sprint
Product Owner agreed to have continuous slack sprint, after every third standard sprint. This sprint is called as Global technical debt sprint (also debt sprint). During debt sprint, all scrum teams i.e. TD team and feature team picks up only internal debt stories. TD team used to coordinate, support and review debt resolution from feature teams. Debt Sprint starts with knowledge sharing and training sessions by TD Team to feature teams, followed by planning for debt sprint, resolving debt stories and end with sprint reviews. Most of design and architectural smells were resolved in debt sprints.

Slack sprint helped us in synergizing complete project team for achieving debt stories, spreading debt awareness, knowledge sharing and synchronizing releases for feature and debt stories.

Slack Sprint continues even after TD team was dissolved. We used slack for continuously refactoring design and coding smells.

4.4 Validate
Validation is the fourth and last stage of four stages. We focus following practices in this stage:

- Validating that, the debt stories are completed as per defined DOD
- Continuous validation which ensured that completed debt stories is not reoccurred
- Validating that, the development teams have not used shortcut or dirty way e.g. copy-pasting of code, to resolve technical debt for sake of proving high business value in less time [Sec. 4.3.2.2]
- Validating that, completing debt stories have not injected other technical debt
- Verifying and validating that, functional workflows have no adverse impacts due to completed debt stories.
- Sharing debt validation results and progress with stakeholders.

Every sprint, testers ensured that all affected workflows are automated. These automated test scripts helped to detect unknown errors as well as validating debt change in less time.

Automated functional test cases and tools [Sec. 4.1] helped in validating above points.

5. RESULT
In this section, we will be showing the impact of presented lifecycle approach in GCP project. We will be taking three examples. Example 1 shows impact on debt due to quality & user defects. Example 2 shows impact on debt due to static analyzer errors. Example 3 shows impact on debt due to memory and GDI leak. Finally, we will close this section by showing results in scaling up users.

5.1 Example 1: Impact on Quality and End User Defects

Table 4 shows that frequency of quality and user defects logged due to technical debt has been reduced considerably from the year 2012 to 2015. Improvement shown here indicates that pragmatic lifecycle approach worked in GCP.

Table 4: Quality and users technical debt defects

Year	**Memory**	**Performance**	**Crash**	**Other Technical Debt**	**Total Technical Debt**
Till 2012	70	165	483	48	766
2012	61	36	132	10	239
2013	10	26	79	1	116
2014	6	16	76	6	104
2015	4	11	20	3	38

5.2 Example 2: Impact on High Priority Static Analyzer Errors

We had a nearly 80-100K code change every year in C++, Java and dot net languages. Life cycle approach and practices mentioned in the paper have helped to reduce static errors despite having continuous code changes. This approach also shows that our practices are very effective in preventing technical debt to last long. Table 5 shows Static analyzer results. The development team has resolved other errors e.g. thousands of Coverity errors, as part of continuous repaying debt as mentioned in step 5 of Execution stage.

Table 5: Prioritized static analyzer errors

	Findbugs	Coverity	FxCop
Prioritized Backlog	123	1107	800
Current Situation	27	400	200

5.3 Example 3: Impact on Memory and GDI Leak

Along with memory and performance errors shown in example 1, we had analyzed all business critical workflow and noticed that nearly all workflows had memory and GDI leak.

Development team executed these workflows in a loop for few hundred times through automated test case and technical debt tools like Purifier, Perfmon and GDI Handle/View identified memory and GDI leak. Table 6 shows improvement in Memory and GDI leak for prioritized workflows.

Table 6: Memory & GDI Leak in GCP

Prioritized Workflows	Memory Leak	GDI Leak
Number of Backlog Stories	26	11
Current Situation	0	0

Table 7 shows Memory and GDI leak in one of the workflow have improved significantly.

Table 7: Memory & GDI Leak in One Workflow

A workflow	Memory Leak (700 Iterations)	GDI Leak (300 Iterations)
Backlog Leak	427 MB	2793
Current Situation	109 MB	0

5.4 Scaling up Number of Users

It was in Year 2012 when Product Manager raised concern that they have planned to scale up number of users from nearly three to four thousand users. Due to an unstable product, they had fear to take the product to new users.

Our pragmatic approach has helped product manager to scale up users not only up to 4K but also beyond 5K users. Figure 4 shows how users have been scaled up in after the year 2012.

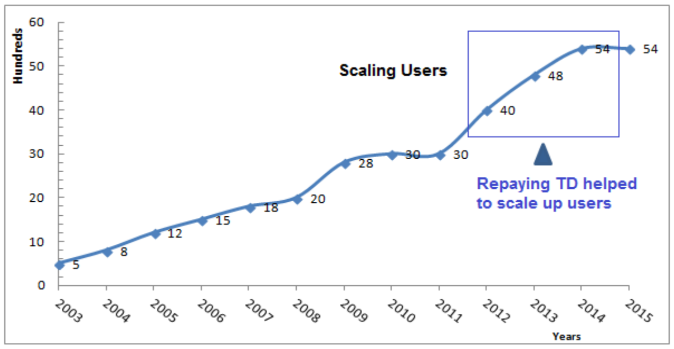

Figure 4: Scaling up users

5.5 Maximizing ROI

Scaling up users definitely contributed in increasing sales and profits for product manager. These practices also contributed towards improving product quality in terms of injecting new defects. As per our study of this financial year, nearly 70% less support calls are reported, 60% less defects escaped into field. It is equivalent of saving nearly two thousand man days of effort. No of repeating testing also reduced drastically which enabled us to reduced release cycle from four months to every month.

6. SUMMARY

We have demonstrated here a case study of GCP project for using pragmatism in our approach for managing technical debt. We had four steps in managing debt. Pragmatic approach suited us along with agile scrum. We were able to resolve continuously prioritized technical debt and bad smells. The pragmatic approach helped us, not only in repaying the existing technical debt, but also assisted in preventing the technical debt to accumulate. We used practices like slack in every iteration and slack iteration for keeping a balance between repaying technical debt as well as continuously integrating new business user stories. As a result, we resolved all prioritized technical debts and the product manager has scaled up more than four thousand users.

We are continuously evaluating new tools that can help us in tackling technical debt in a much more efficient way. For example, we dropped PCLint and started using Coverity [7] since year 2014 because PCLint has vast number of FALSE POSITIVE errors. In comparison Coverity was able to detect not only coding errors, but also runtime errors that had potential to break live system. Coverity helped enormously in improving our product stability.

Currently, we are evaluating tools like 'inFusion' [11] and 'Designite' [10] for design smells as well as sonar cube [28] for better debt dashboard.

7. REFERENCES

[1] Agile Record, Managing Technical Debt with Agile, By Nita Andansare, 25-Sep-2014. http://www.agilerecord.com/managing-technical-debt-agile

[2] AGILE WEB DEVELOPMENT & OPERATIONS, How to translate “business value” of things that are technically

important, by Matthias Marschall on April 19, 2011. http://www.agileweboperations.com/how-to-translate-business-value-of-things-that-are-technically-important

[3] Alistair Cockburn , Blog, Information Radiator, 19-Jun-2008, http://alistair.cockburn.us/Information+radiator

[4] Apache JMeter, Test Performance Tool http://jmeter.apache.org/

[5] Clean Coder, Uncle Bob Consulting LLC, A Mess is not a Technical Debt, Posted by Uncle Bob on 09/22/2009. https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt

[6] CODOVATION, Effective Steps to reduce technical debt: An agile approach, JUN 21ST 2012,http://www.codovation.com/2012/06/effective-steps-to-reduce-technical-debt-an-agile-approach/

[7] Coverity, Coverity Resource Library, Coverity http://www.coverity.com/library/pdf/CoverityStaticAnalysis.pdf

[8] Coverity: Software Testing and Static Analysis Tools http://www.coverity.com

[9] Cppcheck - A tool for static C/C++ code analysis http://cppcheck.sourceforge.net

[10] Design Smell Tool: Designite. http://www.designite-tools.com/

[11] Design Smell tool: inFusion http://www.intooitus.com/products/infusion

[12] DSDM Consortium, Driving Strategy Delivering More, http://www.dsdm.org/

[13] EclEmma - Java Code Coverage for Eclipse http://eclemma.org/

[14] Estimated Interest, by Martin Fowler, 10-Dec-2008. http://martinfowler.com/bliki/EstimatedInterest.html

[15] FindBugs - Find Bugs in Java Programs http://findbugs.sourceforge.net/

[16] Girish Suryanarayana ,Ganesh Samarthyam, Tushar Sharma, Refactoring for Software Design Smells: Managing Technical Debt

[17] InfoQ, Pragmatic Technical Debt Management, Tushar Sharma, Techie, Researcher, Consultant, and Author at Siemens R&D, Sept 25, 2015 http://www.infoq.com/articles/pragmatic-technical-debt

[18] Jason Breault – Agile Expectations, Scrum Team Dependencies, MAY 11, 2015 JASON BREAULT. https://jasonbreault.wordpress.com/2015/05/11/scrum-team-dependencies

[19] Martin Fowler, Technical Debt, by Martin Fowler, 1 October 2003. http://martinfowler.com/bliki/TechnicalDebt.html

[20] Martin Fowler, TechnicalDebtQuadrant, Martin Fowler, 14-Oct-2009 http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[21] MindTools, Mind Tools, Management Training and Leadership Training, Online, Eisenhower's Urgent/Important Principle, Using Time Effectively, Not Just Efficiently, http://www.mindtools.com/pages/article/newHTE_91.htm

[22] NCover - .NET Code Coverage for .NET Developers https://www.ncover.com/

[23] On Technical Debt, Pragmatic Agile Development: How to develop fast & contain Technical Debt, BY ROD NEWING on 06-Jan-2012.http://www.ontechnicaldebt.com/blog/pragmatic-agile-development-how-to-develop-fast-and-contain-technical-debt/

[24] Perfmon, Performance Monitoring Tool http://perfmon.sourceforge.net/

[25] Programmer 97-things: About, The Boy Scout's Rule, by Uncle Bob, 24-November 2009. http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule

[26] Simian - Similarity Analyser | Duplicate Code Detection www.harukizaemon.com/simian/

[27] Slide Share, Identifying and Managing Technical Debt, Dr Nico Zazworka, Dr Carolyn Seaman, 12-Jun-2012.http://www.slideshare.net/zazworka/identifying-and-managing-technical-debt.

[28] Sonar Cube, Evaluate your technical debt with Sonar, By Olivier Gaudin on June 11, 2009. http://www.sonarqube.org/evaluate-your-technical-debt-with-sonar/

[29] SourceMonitor V3.5 - Campwood Software www.campwoodsw.com/sourcemonitor.html/

[30] Sticky Minds, 10 Thoughts on Technical Debt, Volume-Issue: 2011-06, By Matthew Heusser 21-11-2011. https://well.tc/qwy

[31] The Art of Agile, James Shore, I work with people who want to be great. jshore@jamesshore.com http://www.jamesshore.com/Blog/Slack%20and%20Scheduling%20in%20XP.html

[32] The DSUI Team Blog, Debugging a GDI Resource Leak David Wooden - MSFT, 23 Apr 2013, http://blogs.msdn.com/b/dsui_team/archive/2013/04/23/debugging-a-gdi-resource-leak.aspx.

[33] Working Effectively with Legacy Code 1st Edition by Michael Feathers

[34] Zengyang Li,Paris Avgeriou , Peng Liang: A systematic mapping study on technical debt and its management