International Journal of scientific research and management (IJSRM)
\||Volume||3||Issue||5||Pages|| 2880-2882||2015||
Website: www.ijsrm.in ISSN (e): 2321-3418

Improving an Automation Framework by Reducing Technical Debt

Pradeep kumar R S[1], Chethana R Murthy[2]

1R. V. College of Engineering, VTU University
Information Science and Engineering Department
Bangalore
Pradeepkumar.rs484@gmail.com

2R. V. College of Engineering, VTU University
Information Science and Engineering Department
Bangalore
chethanamurthy@rvce.edu.in

Abstract: ***Technical debt is an important analogy pointing to the eventual consequences of poor system design, software architecture and software development within a code base. Technical debt is the cost of programming options and opinions that were assumed consciously to meet a business objective, unknowingly because of lack of knowledge or experience or historically because they made feel originally but are no further best practices today. Technical debt is an old dilemma that raises business risk and cost to the company. All development will result in some amount of technical debt – the objection is to control it, curtail it and establish practices to keep it stabilize that does not impact performance and availability of the critical business services. The proposed study suggests that the software product and process with an eye towards the quantitative definition of the technical debt.***

Keywords: Technical Debt, Refactoring, Deficit Programming.

1. Introduction

Technical Debt is an analogy to describe the cost of making sub-optimal technical design and implementation choices in software product in exchange for releasing the software at a given time. Finding a quantitative measure for technical debt could aid software developers and management in deciding upon courses of action during software development that produce improved project outcomes. Another term for technical debt is 'deficit programming'. Determining that the concept is too vague to obtain general quantitative measures would also be useful. The objective of this paper is to identify quantitative measurements for its definition and use.

Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution. Technical debt is commonly associated with extreme programming, especially in the context of refactoring. That is, it implies that restructuring existing code (refactoring) is required as part of the development process. Under this line of thinking refactoring is not only a result of poorly written code, but is also done based on an evolving understanding of a problem and the best way to solve that problem. Technical debt may also be known as design debt.

The analogy is appropriate inspector often make mindful opinions to deliver new business functionality as early as possible. As with financial debt, they assess the perks of faster time-to-market and expanded yields against the probability of sub-optimal code. Whatever the cause, whether it is in service of cultivating a ambitious edge or meeting conformity requirements, technical debt will arise whenever corners have to be chop in design, coding and testing.

2. Types of Technical Debt

Technical debt comes in a variety of form; whenever an application is initially designed the most significant type comes. Often, the entire outlook of a business service is incompletely understood at design time, so while the design might be ideal in the initial stages of implementation, it may not fit to the many revisions required as the application matures. In other cases, faulty design can be the result of a misconception between the architects, the business and the development team. Still another cause could be stipulation made as an agile project evolves.

Code debt forms a number of issues into one bucket. The most apparent is clumsily written code. This happens primarily because of coding inexperience. Highly complex code may work very properly, but when it is complex, updates to it without a clear understanding of the complexity may result in problems.

Some of the abrupt areas of technical debt come from lack of documentation, lack of process or understanding, lack of building loosely coupled components, lack of collaboration, parallel development, delayed refactoring, lack of alignment to standards, lack of knowledge and lack of testing protocols. Code filled with these kinds of problems becomes fragile. It hardens; making changes to older code can be difficult without the risk that it will break.

3. Properties of Technical Debt

This section analyzes the definition of technical debt. The measuring aspects of the code are not sufficient for evaluating technical debt. Notably the programmer's time,

the 'ideal' of quality and the value of early delivery must all be considered. The cost of programmer time depends on many dimensions including organization, geography, skill level, experience level, environment, and rarity of skills, but for a given project and organization there is typically a small, known, range of values. Programmer time can readily be translated to monetary cost in a given organization.

There are new metrics that reflect how much effort is required in order to get a perfect score on the various axes as shown in the figure below.

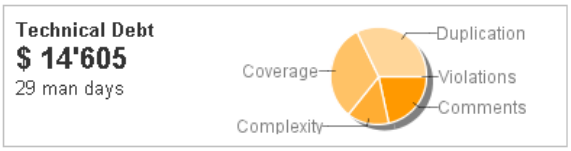

Figure 1.Technical debt metrics.

uses the following formula to calculate the debt :

Debt(in man days) = (cost for fixing duplications) + (cost for fixing violations) + (cost for comment public API) + (cost for fixing uncovered complexity) + (cost for bringing complexity below threshold)

Where :

Duplications = (cost for fixing on block) * (no. of duplicate blocks)

Violations = (cost for fixing one violation) * (mandatory violations)

Comments = (cost for commenting one API) * (public undocumented API)

Coverage = (cost for covering one of complexity) * (uncovered complexity by tests) (80% of coverage is the objective)

Complexity = (cost for splitting a method) * (function complexity distribution >= 8) + (cost for splitting a class) * (class complexity distribution >= 60)

It appears that a key property of technical debt is the above-mentioned 'ideal' to which the software must (eventually) conform. In a practical sense this refers to the knowledge embedded in documents such as coding standards, organizational policies, and design and architecture handbooks and in the knowledge of the developers, managers, architects, and designers working on the project. In an abstract sense, this could be viewed a micro economic 'production function' applied at a more granular level than the usual firm level.

Measures of Technical Debt

Measurement provides the foundation of data upon which analysis, theories, and predictions can be built. In examining the properties of technical debt several units of measurement have been observed. This section surveys potential units of measure for each of these properties. The definitions here lie in the vague middle ground between concrete examples and general axioms, but the goal is to identify the kinds of measures suitable for each property.

The 'value' is defined as "the economic difference between the system as it is and the 'ideal' system", it is necessary to account for both the expense saved and the benefit gained by not conforming the system to its ideal. Expenses and benefits can be expressed in currency units, but they have different sources. The expense can be measured in terms of human effort on the part of the organization. Benefits can also be measured on these terms, but this does not account for the benefit obtained by the use of the software as this use often goes beyond the bounds of the development organization.

This paper restricts itself to measures of debt internal to the development organization. The primary driver of internal costs is programmer time, which can be translated to currency units. There are many other expense components, including hardware, licensing, and the support structures required for developers, managers, executives and other employees.

4. USES OF TECHNICAL DEBT

Technical debt-like measures have been applied to refactoring decisions, scheduling of feature development, project and product quality assessment, development speed, effort estimation for development and maintenance and resource selection. All of these uses are part of the domain of software maintenance. Software maintenance examines the factors involved in making changes to software systems over time. If metrics measure the value of technical debt, software maintenance speaks to its impact, and repayment.

The ISO standard for software maintenance characterizes maintenance as 'corrective', 'preventative', 'adaptive', or 'perfective'. Corrective maintenance is the modification of software to correct a discovered problem after its release. Preventative maintenance is the modification of software after its release to prevent a problem from occurring. Adaptive maintenance is the modification of software to allow it to conform to changes in its hardware or software environment. Perfective maintenance is the modification of software to correct latent faults, whether they affect program behavior, documentation or maintainability.

This framework calls for the creation of a "technical debt item" record for each discovered piece of technical debt. Each item is assigned a description, a date recorded, a person responsible, a component location, and a type, which reflects the project phase the debt, is incurred in. Each item has attributes of principal, interest amount and interest probability assigned an ordinal value of 'low', 'medium', and 'high' to reflect a coarse-grained notion of the item's debt impact. These estimated values are then refined through the use of historical data from the organization and the project as it proceeds. The goal of the framework is to support project-level decision-making, to provide reference data for future projects, and to validate the proposed framework.

5. LIMITATIONS

The papers examined are a tiny proportion of the papers published on the topics of software quality, maintenance, cost estimation, and software metrics. Even within the span of the papers surveyed there is too wide arrange of dimensions, metrics, and values across too wide a span of concerns to be hopeful of being precise. While the aim of the paper is to serve as an introduction to the literature around technical debt, this cannot hope to be a thorough survey of the field, given the wide range of topics addressed.

6. Conclusion

During the course of this paper, we discussed that the part of continuous service and process improvement, it can be crucial to organize peer review of code as well as enforcing a development mentor program to assure carry on knowledge transfer. Review for ways to document code either allowing it to the responsibility of the developer or by accomplished using software tools. And positively, appreciate that the small boost in time required for a quality software project will be in addition to pay for itself in reduced technical debt, both principle and interest.

.

References

[1] W. Cunningham. The WyCash portfolio management system. *Addendum to the Proc. on Object-Oriented Programming systems, languages, and applications*. Pp 29-30. 1992.

[2] N. Brown et al. Managing Technical Debt in Software-Reliant Systems, *FSE Workshop on Future of Software Engineering Research*. Pp 47-52. 2011.

[3] B. W. Boehm. Software engineering economics. *IEEE Trans. Software Eng. SE-10(1)*, pp. 4-21. 1984.

[4] B. Kitchenham, What's up with software metrics? – A preliminary mapping study, *Journal of Systems and Software 83(1)*. 2010.

[5] A. Meneely, B. Smith, and L. Williams, "Software Metrics Validation Criteria: A Systematic Literature Review.", *Trans. on Software Engineering and Methodology*, to appear

[6] A. Israeli and D. G. Feitelson. The linux kernel as a case study in software evolution. *J. Syst. Software 83(3)*, pp. 485-501. 2010.

[7] M. Agrawal and K. Chari. Software effort, quality, and cycle time a study of CMM level 5 projects. *IEEE Trans. Software Eng. 33(3)*, pp. 145-56. 2007.

Author Profile

Pradeep kumar R S currently pursuing M.tech in Information Technology in R.V.College of Engineering, Bangalore**.**