At the Intersection of Technical Debt and Software Maintenance Costs

Ward Cunningham introduced the notion of technical debt in 1992, writing "Shipping first time code is like going into debt. A little debt speeds up development so long as it is paid back promptly with a rewrite". [1] Technical debt can be described as a quantification of the amount of 'should fix' issues that remain in production code. The term "technical debt" is a metaphor used to ease business leaders understanding of the costs of making poor decisions during development or making wise decisions during development (such as a short cut to meet a time to market goal) and not following up with a quick rewrite to address the short cut. It is intended to facilitate discussions between business leaders and the software development team. The notion of technical debt should help create a situation where the development team accepts the fact that certain business realities will sometimes create an environment where best practices and processes must be thrust aside and in concert with this, the business leaders understand that decisions to incur this 'debt' are only wise if there is also a plan in place to return the code to a proper state as soon as possible.

Technical debt is not the same as software quality, though there is certainly an intersection between software quality measures and characteristics that indicate technical debt. Software quality measures include indications of how well the software meets user requirements along with how well those requirements are delivered. Technical debt speaks to the structural quality of the software. An application that meets all the user requirements with a completely satisfactory user interface could be sitting on a mountain of technical debt if programming practices or coding standards were detoured, or methods and functions were poorly documented. Tools and methodologies that measure technical debt look for things such as uncommented methods, artifacts with high internal complexity, use of global variables, artifacts with high fan-out, duplicated code, inadequate test coverage, etc. Even an application developed using all best practices could amass technical debt over time if it is not evolving to keep up with current technology.

Clearly assessing the cost of addressing technical debt is not the same thing as determining the cost of maintaining a piece of software. There is however strong evidence that an understanding of an application's technical debt potential should be an important piece of information when an estimate is being performed for the maintenance of that application. Much of what is considered technical debt speaks specifically to the things that will make the software harder to maintain and upgrade. The intent of this paper is to explore this hypothesis further. It is a report on an on-going research project. The first section discusses why debt is a good analogy for the software world. In the second, the definition of technical debt is refined and the different types of classifications of technical debt are discussed. This is followed by a discussion of methods for measuring technical debt for an existing application or code base. The next session discusses software maintenance and speaks to areas where software maintenance and technical debt intersect. The fifth section discusses where technical debt analysis and understanding can inform better software maintenance estimates. The final section wraps up the paper and discusses future work on this research project.

Why Debt?

We all understand that sometimes assuming a debt is reasonable and even wise. If you were considering starting a web development and hosting service, you would want to ensure that you had adequate

resources to deliver satisfactory solutions to your clients. Unless you are independently wealthy you will probably need to borrow money from the bank or some other financial institution. If you have done a good job researching the market you intend to focus on and have identified that the demand for your service will be high – this investment seems like a wise choice. As you begin to service clients, you will be able to pay the principal and the interest with the income you receive. If you are entering this business on a whim, without proper research or understanding, you would be well advised to not borrow money for this endeavor until you are sure that you will be able to generate enough busi ness to pay the loan off in a timely fashion.

Similarly, if you are a software application provider, there may be reasons why assuming some technical debt is wise. If you are in the midst of implementing a new feature that will bring delight to many of your current users when you discover you're competitor is about to release this same feature, you may choose to take some short cuts to get to market with it first. You have made the conscious decision to abandon doing it right because you believe that being first to market will increase revenues. If you have users that are unhappy with the pace at which you are addressing a bug or new feature request, so much so that they may refuse to renew their license, you may decide it is wise to address their specific issues as rapidly as possible to ensure renewal. Once again, this is a conscious decision to divert process requirements to combat a potential revenue loss.

When one refers to technical debt in their software applications, they are referring to the places where shortcuts have been taken. Forms in which these shortcuts might present include:

- Lack of documentation
- Lack of adherence to processes or standards
- Missing tests
- Lack of modularization
- Architecture or design that is not well thought out or scalable
- Failure to keep up with technology

These shortcomings that are introduced into production software can be thought of as the principal of the debt – the cost (and effort) required to address these shortcomings. As with financial debt, there is also interest. These shortcomings will also hamper future development. The maintenance and update of poorly documented code is more expensive than code that is well commented and delivered with supplemental documents. Making changes to software that does not have an established test suite is very risky. Lack of modularization makes it difficult for development or maintenance teams to address new requirements with agility. The original debt, if not addressed promptly, has costs each time the debt ridden code is touched

Technical debt categorizations

Technical debt as a metaphor is different than the technical debt that resides in your code. The metaphor is a tool for communication and negotiation; technical debt in your code refers to the amount of effort associated with addressing the violations (of good practices, standards, etc.) that currently reside in your code, whether they were introduced consciously or through sloth. While the fact that there is technical debt implies that bad code may have been written; in the context of the metaphor, having technical debt is not what happens when inexperience, unsupervised junior developers (or just bad developers) write bad code because they don't know any better. That's just bad code. The results may be the same with

respect to the technical debt in your code, but the metaphor does not apply; this was not a conscious decision to borrow now for short term gain with the commitment to redress soon. Applications with this kind of a history have bigger issues to deal with than rework and updated documentation, they are generally doomed to fail.

Technical debt is also not a measure of the defects in the code. Defects indicate missing or poorly implemented user requirements – the functional quality of the code. Technical debt focuses on the structural quality of the code. In fact, much of the side effects of this technical debt are things that the end user may never see directly. It's not a measure of the things that make the system crash or invoke unusual and unclear error messages from your application or the operating system. What the end user of a system with accumulating technical debt might notice are the degrading performance of the application, dwindling frequency of updates and bug fixes, corrupt data or security breaches.

Fowler classifies technical debt as falling into two dimensions; reckless or prudent and deliberate or inadvertent [3]:

- Reckless and inadvertent – "What's encapsulation?" (this would fall into the bad code category mentioned above)
- Reckless and deliberate – "We don't have time for design"
- Prudent and inadvertent – "Now we know how we should have done it"
- Prudent and deliberate – "We must ship now and deal with the consequences lat er"

Another way to classify technical debt focuses on how it was incurred [4]:

- Unavoidable debt – due to changes in the law or certifications – generally unavoidable
- Strategic debt – usually incurred proactively
- Tactical debt – no time to do it right – reactive rather than proactive
- Incremental debt – lots a minor short cuts that add up
- Naïve debt – back to bad code as mentioned above

How technical debt is measured

Clearly, here the reference is to the technical debt in your code, not the metaphor. From a measurement perspective the units could either be the labor hours (or currency) required to address technical debt or some proxy of these labor hours (such as story points, function points, etc.) There are several approaches an organization can employ to measure technical debt:

- Organizations who embrace the metaphor should have an excellent idea of the technical debt incurred intentionally. Presumably the discussion between the team and the business leaders includes a discussion of the return on investment of incurring the debt, which most likely includes disclosure from the team that meeting a short term requirement will result in X amount of rework (generally quantified as Story Points, Function Points, effort hours, etc.)
- Some organizations use various metrics such as code coverage, number of tests per size measure, coupling or cohesion factors, cyclomatic complexity, average function or method size, number of comment lines per line of code, etc. Many development environments will provide some or all of these measure for the code base. These types of measures will help an organization determine how much technical debt exists and where they are likely to optimize the time spent with rework

- There are also tools available – both commercial and open source that perform static code analysis. These tools scan the code to look for violations. The rules for assessing such violations are drawn from software engineering standards and literature, resources such as the Common Weakness Enumeration (cwe.mitre.org) website, a free resources of common software weaknesses, and through organizations such as the Consortium for IT Software Quality (CISQ), Object Management Group (OMG) and the Software Engineering Institute (SEI). These tools provide an excellent way for an organization to identify the technical debt (not the metaphor) that has worked its way into their code.

According to Curtis, et.al. [5],[6], [7] technical debt is determined by examining the following quality aspects of your code:

- Robustness - an indication of the stability or resilience of an application. How well it avoids outages and how quickly it can recover when they occur. Violations that indicate lack of robustness include:
    - Uncontrolled data access
    - Poor memory management
    - Empty catch block
    - Open resources not being closed
- Performance efficiency – an indication of the applications speed and the efficiency with which it uses resources. Violations that indicate lack of performance efficiency include:
    - Large indices on large tables
    - Complex queries on big tables
    - Static vs. pooled connections
    - Expensive operation in a loop
- Security – an indication of how well the application is able to prevent unauthorized intrusions and protect application data. Violations that indicate security problems include:
    - SQL injection
    - Cross-site scripting
    - Buffer overflow
    - Uncontrolled format string
- Transferability – an indication of the ease with which a new team can acclimate and understand the application and how quickly they can become productive working with it. Violations that indicate an application may not be transferable include:
    - Unstructured code
    - Misuse of inheritance
    - Lack of comments
    - Violating naming conventions
- Changeability – an indication of the ease with which modification can occur without the injection of new defects. Violations that would indicate reduced changeability include:
    - Highly coupled component
    - Duplicated code
    - High cyclomatic complexity

Intersection of technical debt with software maintenance

Robert Martin [8] says "It doesn't take a lot of skill to get a program to work. The skill comes in when you have to keep it working." Software maintenance is generally considered to include any software engineering activities that are performed on a software application after it has been released into production. There is a fairly common misperception that software maintenance only involves fixing defects; in actuality much of what falls under the classification of software maintenance involves activities associated with adding new functionality, updating for technology or making the software easier to maintain. Software maintenance is generally divided into the following classifications:

- Corrective – tasks associated with fixing defects
- Preventive – tasks associated with addressing problems in the code that are not currently causing bad behavior but have the potential to make the code error prone or hard to maintain
- Adaptive – tasks associated with making sure that the software continues to thrive as hardware and technology progress
- Perfective – tasks associated with adding new features and capabilities[1]

It seems that on average, approximately 80% of maintenance efforts go towards activities not associated with defect correction. If one were to think about the different types of maintenance using the definitions above, it seems that preventive and adaptive maintenance are specifically focused on eliminating technical debt while corrective and perfective maintenance are focused on making changes to features and how those features are delivered. But even changes associated with corrective and perfective maintenance are impacted by technical debt because according to the Guide to the Software Engineering Body of Knowledge (SWEBOK Guide) 40-60% of most maintenance tasks are spent trying to understand the software being maintained.

Clearly a significant cost driver in any software maintenance exercise should relate to those factors that indicate technical debt. And while static analysis tools give a great deal of insight into the amount of technical debt in an application – they are expensive to use; even the good open source tools require a significant amount of learning and configuration. There are however some metrics that may be useful to establish a proxy for technical debt that are often available directly from the software development environment (many are also available as outputs from the open source unified code counting tool from the University of Southern California Center for Software Excellence (USC CSE)[2]:

- Software size in Source Lines of Code (via automatic code counting mechanism) – while SLOC is not necessarily a primary driver in a maintenance effort, it is generally well accepted that the larger the software application the more complex it will be to understand the code and find and fix errors, while avoiding complexities
- Class Coupling is a measure of how many classes a single class uses – also referred to as Coupling between Objects. The higher this number the harder it will be to make changes without having unwanted side effects. This metric only makes sense for applications using object-oriented technology.

[1] It is important to note that while these classifications and their definitions are common among industry experts, not all experts use the same terminology, and not all agree with these definitions

[2] Available for download at http://sunset.usc.edu/ucc_wp/

- Cyclomatic Complexity is a measure of the amount of decision logic (conditionals, branching, etc.) in the code. The more decisions that need to be made the harder the code is to follow and the more testing is required to identify unintended consequences of a change.
- Halstead Volume is a measure that assesses code complexity and maintainability by examining the number of unique operators (+,-,/, etc.) and operands (literals) compared to the total number of operators and operands in a software program using this comparison as a proxy for complexity. The more voluminous the software, the harder it is to understand and manage change.
- Average Depth of Inheritance indicates the length of the inheritance tree from node to root. While more inheritance may signal an effective application of reuse, it can also lead to code that is hard to follow. This metric only makes sense for applications using object-oriented technology.
- Average Percent of lines of comment per module or function indicates what percent of the code is commented indicating how easy it would be to transfer or change the code (this measure of course cannot measure the quality of the comments so it relies on some good faith assumption that developers are commenting well)
- Average Size of Functions or classes measured in source line of code. Smaller chunks of code are much easier to understand and maintain than larger ones which tend to do multiple things that may be better refactored into several smaller chunks.
- Maintainability Index – this is a metric that was introduced in 1992 and later refined in a paper in IEEE Computer in 1994 [12]. The original version of the metric was based on Halstead Volume, Cyclomatic Complexity and SLOC, while a later version introduced by the SEI included percent of lines of comments as well. This metric is calculated by some modern software development tools including Microsoft© Visual Studio (since 2007), JSComplexity and Radon.
- Number of Tests per function, class or module – The more tests there are, the easier it will be to make changes which don't have unintended consequences, making the software more maintainable.

These are just a few of the many different methods used to determine the complexity and maintainability of software [13]. What's attractive about these particular measures is that there are many tools available that routinely collect at least a subset of these values. Many of these tools also have facilities to track effort against modules or classes or functions, so an organization developing software can easily perform studies determining which of these measures seem most likely to impact software maintenance effort and where to focus rework to optimize its value. And while opinions in the literature vary widely as to the efficacy of these measures for overall software maintenance effort prediction, they should at least be considered as part of the equation when determining understandability of existing software packages. This author is currently conducting on-going data collection and analysis on this topic, though not enough data has been collected to date to draw conclusions.

Estimation Considerations for Software Maintenance

There are several ways to think about the estimation of software maintenance costs. Depending on the types of maintenance that are being accomplished different tasks need to be accomplished. Table 1 depicts the various activities.

Type of Maintenance	Tasks
Corrective	
	Code Understanding
	Problem Identification and Analysis
	Design
	Implementation
	Regression Testing
	Acceptance Testing
Perfective	
	Code Understanding
	Requirements Analysis
	Design
	Code and Unit Test
	Software Integration and Test
	Acceptance Testing
Preventive and Adaptive	
	Code Understanding/Reverse Engineering
	Problem Identification and Analysis
	Forward Engineering
	Implementation
	Regression Testing
	Acceptance Testing

Table 1: Software Maintenance Tasks

As mentioned earlier, the corrective and perfective types of maintenance are quite similar in the activities that are being performed (Problem Identification and Analysis for a defect is quite similar to requirements analysis for a new feature – the distinction is mostly that it is likely that different types of resources would be attempting these tasks), and in fact these maintenance types should be handled from an estimating perspective in a manner similar to estimating a new software development. The question then becomes, how does one address the issue of code understanding and development or maintenance team unfamiliarity. Here we will use Constructive Cost Model from University of Southern California Center for Software Excellence (USC CSE COCOMO II) as an example, as it is an open model with algorithms that can be discussed freely. This methodology and recommendations discussed here should apply equally to other software cost estimation models.

The COCOMO II model has a software maintenance model, which seems mostly suited to perfective and corrective maintenance. This model is quite similar to the devlopment model with several exceptions;

- Size = Code Base Size * Maintenance Change Factor (MCF) * Maintenance Adjustment Factor (MAF) where:
    - MCF is a user input that can be determined by (Size Added * Size Changed)/Total Code Base
    - MAF is 1+(Software Understanding * Programmer Familiarity) where…

    - Software Understanding is a user input value from 10 to 50 that indicates structure, application clarity and self-descriptedness (very aligned with the Maintainability Index)
    - Program Familiarity describes the team familiarity with the code
- SCED (schedule compression) and RUSE (Design for Reuse) are no longer cost drivers in the maintenance mode because both were deemed to have significant impact on cost of maintenance projects
- RELY (Reliability) has the Reverse effect than it does during development as software that is developed for high reliability should have fewer defects and be easier to update.

In the case of the COCOMO II model, issues associated with maintainability and development or maintenance team familiarity are used to influence the 'size' of the maintenance job. There is no specific activity for Code Understanding but rather it is weaved throughout all the activities through its impact on size. By insisting on a value no less than 10, the COCOMO II model enforces the notion that even very maintainable software is going to see an increase in effort simply devoted to figuring out what is happening in the code. Other software maintenance models have similar facilities for adjusting maintenance 'size' through factors so a similar methodology can be applied. The estimator should use caution with Development or Maintenance Team Familiarity; in the COCOMO II Maintenance model – this input only effects maintenance size, if you are using a model that uses this input as a cost driver across the maintenance activities, including it into the size adjustment as well could result in double dipping. An alternative to incorporating code understanding factors into the maintenance 'size' would be to create productivity adjustments to each activity to adjust for code understanding activities. The benefit of this would be that some activities, such as requirements and design are more impacted by the understanding of the code, while activities associated with test may be less impacted by these factors and more impacted by the number and coverage of existing tests.

Issues associated with adaptive and preventative maintenance types are a little less clear cut. While the notion of adjusting maintenance 'size' by factors indicating maintainability and team familiarity still have merit, the issue of determining size is a bit more complicated. For Corrective and Perfective maintenance, one knows the size of the Code Base (because it is completed software application where SLOC or Function Points can be counted using existing standards or automatic code counting tools) and one knows how much functionality will be changed and how much functionality will be added. While estimating size is never an exact science, the same rules apply for these types of maintenance projects as are applied when estimating new code development. When the changes are not directly associated with changes in functionality, the sizing of the maintenance job is further complicated. Here's an area where understanding technical debt could really become useful. Because the causes of technical debt are not directly customer facing, the development or maintenance team needs to take on a decision to address a certain amount of technical debt within a specific release. Once this decision has been made, there must be a determination of what the most optimal application of this allotted effort would be. The development team should be able to help the estimator align their rework goal with the functionality that would be touched. This discussion should be facilitated through metrics available via development tools or static code analysis tools. This establishes a size proxy for the rework allocation for the current release. If the rework is planned for areas where functionality is changing as well – care should be taken to ensure

that any overlaps are accounted for. Once this size proxy has been determined, the maintainability and development or maintenance team familiarity still needs to be accounted for as part of the estimate.

Conclusions and Further Works

The software industry uses the term technical debt in two subtly different ways:

- It is used as a metaphor to describe to business leaders the costs of tradeoffs between good development practices and business requirements
- It is used as a root level indication of the amount of effort required to address all of the violations (of good practices, coding standards, etc.) in a code base or application

Sometimes the discussion of technical debt is muddied by these two meanings. In the former it is assumed that technical debt is acquired consciously and conscientiously, while the later definition includes that technical debt that is incurred through sloppiness and lack of conscientiousness. At some level this is an unimportant distinction, as most technical debt will become problematic if it is allowed to linger to long in the code.

Some technical debt leads to a direct indication of the amount of software estimation required to address it, some just offers an indication of how much additional effort is associated with a change of the code base due to issues in the code that make it hard to understand and follow. Software estimators would do well to have a discussion with the maintenance team prior to estimation to determine what, if any, changes to non-functional aspects of the application are planned for a release. These changes need to be part of the estimate. Regardless of whether the release is intended to address some aspect of technical debt, the estimator should broach the subject of measured and/or perceived understandability of the code to be changed or added to for the release being estimated.

This research effort is on-going. The goal is to work to establish technical debt metrics that prove to be correlated to software maintenance effort (or cost) and to augment existing maintenance cost estimating relationships (or size estimating relationships) with these metrics. It is important to identify metrics that are easy and affordable to count. The COCOMO II model is a good example of how technical debt can be used in assessing the maintainability of existing code when adding or adapting functionality. More work is required to address ways of associating maintenance size for efforts addressing other forms of technical debt (performance, security, robustness, technology) outside the scope of changed user facing functionality.

References

[1] http://en.wikipedia.org/wiki/Technical_debt

[2] "Technical debt 101: A primer about technical debt, legacy code, big rewrites and ancient wisdom for non-technical managers", available at https://medium.com/@joaomilho/festina-lente-e29070811b84 (retrieved 3/12/2015)

[3] Codabux, Z., Williams, B.,"Managing Technical Debt: An Industrial Case Study, 2013, International Workshop on Managing Technical Debt, San Francisco, CA, May 2013, p8-15

[4] Ergin, L., "Technical Debt- Do not Underestimate the Danger", available at http://www.slideshare.net/lemiorhan/technical-debt-do-not-underestimate-the-danger (Retrieved 3/12/2015)

[5] Curtis,B., Sappidi, J., Szynkarski, A., "Estimating the Principal of an Application's Technical Debt", IEEE Software, vol 29, no. 6, pp34-42, Dec 2012

[6] Curtis,B., Sappidi, J., Szynkarski, "Estimating the size, code and types of Technical Debt", Third International Workshop on Managing Technical Debt 2012, 2012

[7] Curtis, B, "Measuring and Managing Technical Debt", available at http://omg.org/news/meetings/tc/tx-14/special-events/cisq-presentations/CISQ-Seminar-2014-6-17-BILL-CURTIS-Measuring-and-Managing-Technical-Debt.pdf, (Retrieved 3/12/2015)

[8] "Software Maintenance", available at sce2.umkc.edu/BIT/burrise/pl/evolution/SoftwareMaintenance.ppt (Retrieved 3/23/2015)

[9] Glass, Robert, "Frequently Forgotten Fundamental Facts about Software Engineering", IEEE Software, May/June 2015, available at http://www.kictanet.or.ke/wp-content/uploads/2012/08/Forgotten-Fundamentals-IEEE-Software-May2001.pdf

[10] Grubb, Penny, *Software Maintenance: Concepts and Practice,* World Scientific Publishing Co, 2003

[11] Guide to the Software Engineering Body of Knowledge (SWEBOK Guide) available at http://www.computer.org/web/swebok/;jsessionid=6ec805fde317d577dcb8bb1058ca#Ref1.1 (Retrieved 3/2015)

[12] Coleman, D., Ash D., Lowther, B., Oman, P., "Using Metrics to Evaluate Software System Maintainability", IEEE Computer, 1994, 0018-9162

[13] Saini,M, Chauhan, M., "A Roadmap of Software System Maintainability Models", International Journal of Software and Web Sciences, USWS 12-359, 2013 available at http://iasir.net/IJSWSpapers/IJSWS12-359.pdf (Retrieved March 2015)