White Paper

Paying Down the Interest on Your Applications

A Guide to Measuring and Managing Technical Debt

For those with responsibility to govern the costs and risks of application portfolios, the financial metaphor "Technical Debt" helps us think about software quality in business terms. This paper includes a formula to benchmark your Technical Debt against industry data, or adjust the parameters to best fit your organization's own maintenance and structural quality objectives, experiences, and costs. It also details the "Technical Debt Management Cycle", a 7-step process for analyzing and measuring Technical Debt so you can relate executive business priorities to strategic quality priorities for reducing business risk and IT cost.

Dr. Bill Curtis
Senior Vice President and Chief Scientist, CAST
Director, Consortium for IT Software Quality

Executive Summary

Dr. Bill Curtis, SVP and Chief Scientist at CAST details the source of Technical Debt in enterprise applications, and provides concrete steps to measure and manage this debt for those with executive responsibility to govern the costs and risks of application portfolios.

The Problem

- Technical Debt is created when developers write software that violates good architectural or coding practices resulting in structural flaws in the code.
- The structural flaws that cause Technical Debt must be reduced to control development costs or avoid operational problems or the cost and risk of the application will grow unacceptably.
- Historically, IT organizations did not have a way to measure or estimate Technical Debt, and therefore it could not be used to estimate the costs of application ownership or to guide management decisions about how much to invest in application quality.
- Trying to make decisions about retiring Technical Debt at a global level can be overwhelming given limited budgets and it is often difficult to visualize the expected payoff.

The Solution

- Choosing ‘debt’ as a metaphor engages a set of financial concepts that help executives think about software quality in business terms.
- Advances in software analysis and measurement allow more accurate estimates of Technical Debt in an application based on actual counts of structural problems, making Technical Debt a critical tool for application management.
- For management to make critical decisions on resource allocation to eliminate violations that comprise Technical Debt, they need to distinguish the type of threat and prioritize the importance of each area.
- The Technical Debt Management Cycle is a 7-step process for analyzing and measuring Technical Debt that relies on relating it to executive business priorities and developing specific targets for each application based on strategic quality priorities with an expectation of the benefit to be achieved for reducing business risk and IT cost.

Executive Summary (continued)

Key Takeaways

- Technical Debt is a risk to the cost and operational performance of critical business applications, and left unaddressed can so degrade an application that its benefits to the business cannot be justified by its growing costs or operational risks.
- CAST has calculated the Technical Debt individually for 745 applications containing 365 million lines of code (11.3 million backfired function points) submitted by 160 organizations, and found an average Technical Debt of $3.61 per line of code. Consequently, a typical application accrues $361,000 of Technical Debt for each 100,000 lines of code, and applications of 300,000 or more lines of code carry more than $1 million of Technical Debt.
- Using the Technical Debt formula presented in this paper, an organization can benchmark against industry data, or adjust the parameters to best fit their own maintenance and structural quality objectives, experiences, and costs.
- This paper also presents details on integrating the measurement and management of Technical Debt into Lean and Agile development environments.

Contents

I. Introduction

Ward Cunningham initiated the Technical Debt metaphor in 1992[1] by referring to violations of good architectural and coding practice as 'debt' (see sidebar). However, the fundamental problem underlying Technical Debt was formulated back in the 1970s by Meir Lehman[2] who posited in one of his laws of software evolution that as a "system evolves its complexity increases unless work is done to maintain or reduce it." This complexity degrades the performance of business applications, increases their likelihood of failure, and multiplies the cost of owning them.

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite…The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation…" - Ward Cunningham

Technical Debt is created when developers write software that violates good architectural or coding practices resulting in structural flaws in the code. Technical Debt represents the cost of fixing structural quality problems in production code that the organization knows must be eliminated to control development costs or avoid operational problems. As shown in Figure 1, the reasons for these violations can vary from simple explanations like an inadvertent coding mistake to intentional shortcuts based on business needs. Cunningham was only referring to structural problems that result from conscious design tradeoffs or coding shortcuts to get functionality running quickly; we embrace a broader approach to include all structural problems that an IT organization prioritizes as 'must fix'.

According to industry luminary Steve McConnell[3], sometimes Technical Debt is an unintentional consequence of inexperience or incorrect assumptions, while in other cases it is intentional, as in Cunningham's definition, in order to get new functionality running quickly. In either case, the development team knows, or ultimately learns, that it has released software with structural flaws that must be fixed or the cost and risk of the application will grow unacceptably.

Highlights

Technical Debt emerges from poor structural quality and affects a business both as IT cost and business risk.

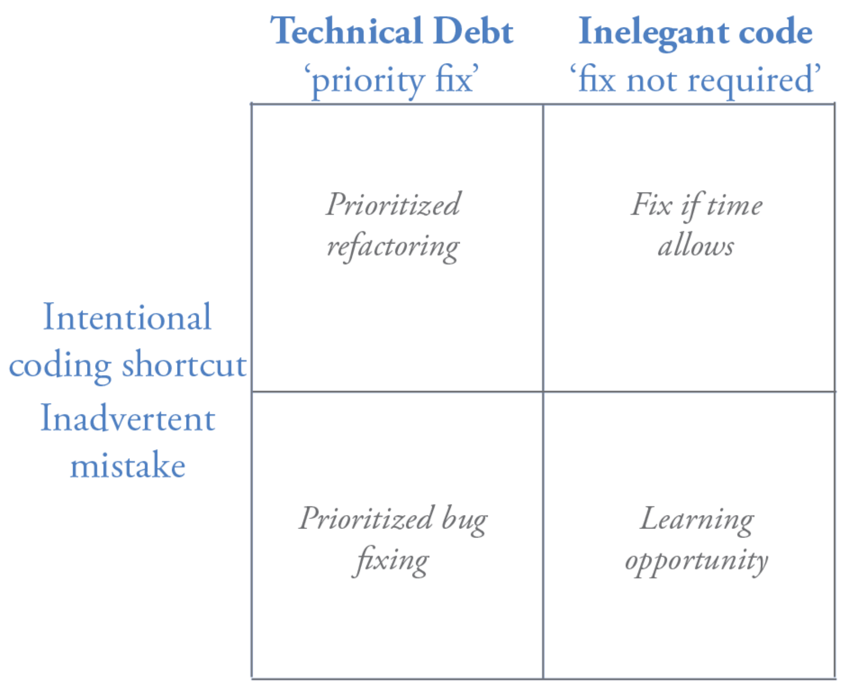

Figure 1. Reasons for Technical Debt

Technical Debt must be distinguished from defects or failures. Failures during test or operation may be symptoms of Technical Debt, but most of the structural flaws creating Technical Debt have not caused test or operational failures. Some may never cause test or operational failures but instead make an application less efficient, less scalable, more difficult to enhance, or more penetrable by hackers. In essence, Technical Debt emerges from poor structural quality and affects a business both as IT cost and business risk.

II. Technical Debt's Affect on the Business

First and foremost, Technical Debt is a risk to the cost and operational performance of critical business applications. Left unaddressed, Technical Debt can so degrade an application that its benefits to the business cannot be justified by its growing costs or operational risks. Consequently, managing Technical Debt is an executive responsibility of those held accountable for governing the costs and risks of application portfolios.

The value of the Technical Debt metaphor has been limited because most IT organizations did not have a way to measure or estimate it. Without measurement, Technical Debt cannot be used to estimate the

Highlights

Choosing 'debt' as a metaphor engages a set of financial concepts that help executives think about software quality in business terms.

costs of application ownership or to guide management decisions about how much to invest in application quality. Advances in software analysis and measurement have solved this problem and have made Technical Debt a critical tool for application management.

III. The Full Technical Debt Metaphor

Choosing 'debt' as a metaphor engages a set of financial concepts that help executives think about software quality in business terms. In this section we will define the concepts required to apply the full Technical Debt metaphor so that each factor can be measured and used in analyzing the structural quality of applications in financial terms. The concepts defined below provide a foundation for the economics of software quality, a topic developed in greater detail by Capers Jones and Olivier Bonsignour[4].

- **Technical Debt**: Future costs attributable to known structural flaws in production code that need to be fixed, including both principle and interest. A structural flaw in production code is only included in Technical Debt calculations if those responsible for the application believe it is a 'must-fix' problem. Technical Debt is a primary component of the cost of application ownership.
- **Principal:** Cost of remediating must-fix problems in production code. At a minimum the principal is calculated from the number of hours required to remediate these problems in production code, multiplied by the fully burdened hourly cost of those involved in designing, implementing, and testing these fixes.
- **Interest:** Continuing costs, primarily in IT, attributable to must-fix problems in production code. These ongoing costs can result from the excessive effort to modify unnecessarily complex code, greater resource usage by inefficient code, etc.
- **Business Risk:** Potential costs to the business if must-fix problems in production code cause damaging operational events such as outages, incorrect computations, lost productivity from performance degradation, and security breaches.
- **Liability:** Costs to the business resulting from operational problems caused by flaws in production code. These flaws include both must-fix problems included in the calculation of Technical Debt as well as problems not listed as must-fix because their risk was underestimated.
- **Risk:** Potential liability to the business if a must-fix problem in production code caused a liability-inducing event. Risk will be

expressed in terms of potential liability to the business rather the IT costs that are accounted for under 'interest'.

- **Opportunity Cost:** Benefits that could have been achieved had resources been committed to developing new capability rather than being assigned to retire Technical Debt. Opportunity cost represents the tradeoff that application managers and executives must weigh when deciding how much effort to devote to retiring Technical Debt.

The relationships in the Technical Debt metaphor are displayed in Figure 2. The cost to fix structural quality problems constitutes the principal of the debt, while the inefficiencies they cause such as greater maintenance effort or excessive computing resources represent interest costs on the debt. The structural problems underlying Technical Debt also create business risks such as outages and security breaches, and the negative events they can cause result in liabilities such as lost revenue from online sales or costly clean-up from a security breach. The effort committed to remediating Technical Debt instead of new business functionality represents opportunity costs related to lost benefits that might otherwise have been achieved.

Highlights

The effort committed to remediating Technical Debt instead of new business functionality represents opportunity costs related to lost benefits that might otherwise have been achieved.

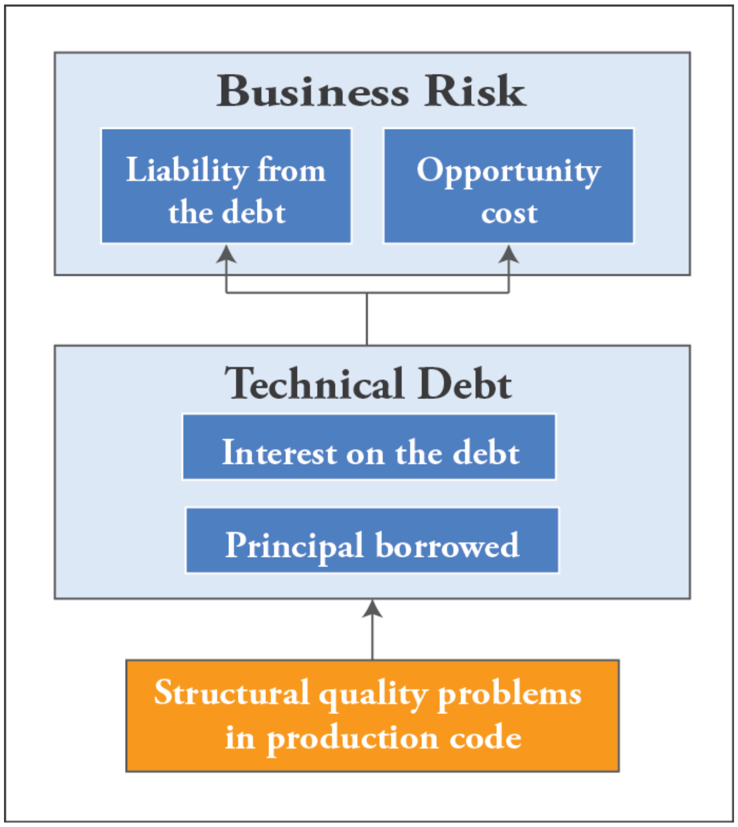

Figure 2. Components of the Technical Debt Metaphor

IV. Measuring Technical Debt

Historically, it has been difficult to determine an exact measure of Technical Debt since its calculation is based only on the structural flaws that the organization intends to fix, including those that may not have been detected yet. However, modern software analysis and measurement technology allows us to more accurately estimate the amount of Technical Debt in an application based on actual counts of structural problems. By analyzing the structural quality of application, rating the severity of each problem, and categorizing the must-fix problems, IT organizations can now automate the measurement of Technical Debt.

Calculating Technical Debt is a function of three variables—the number of must-fix problems in an application, the time required to fix each problem, and the cost for fixing a problem. Each of these variables can be measured or estimated to develop a formula for computing Technical Debt. These variables and their data sources are displayed in Figure 3.

Highlights

Modern software analysis and measurement technology allows us to more accurately estimate the amount of Technical Debt in an application based on actual counts of structural problems.

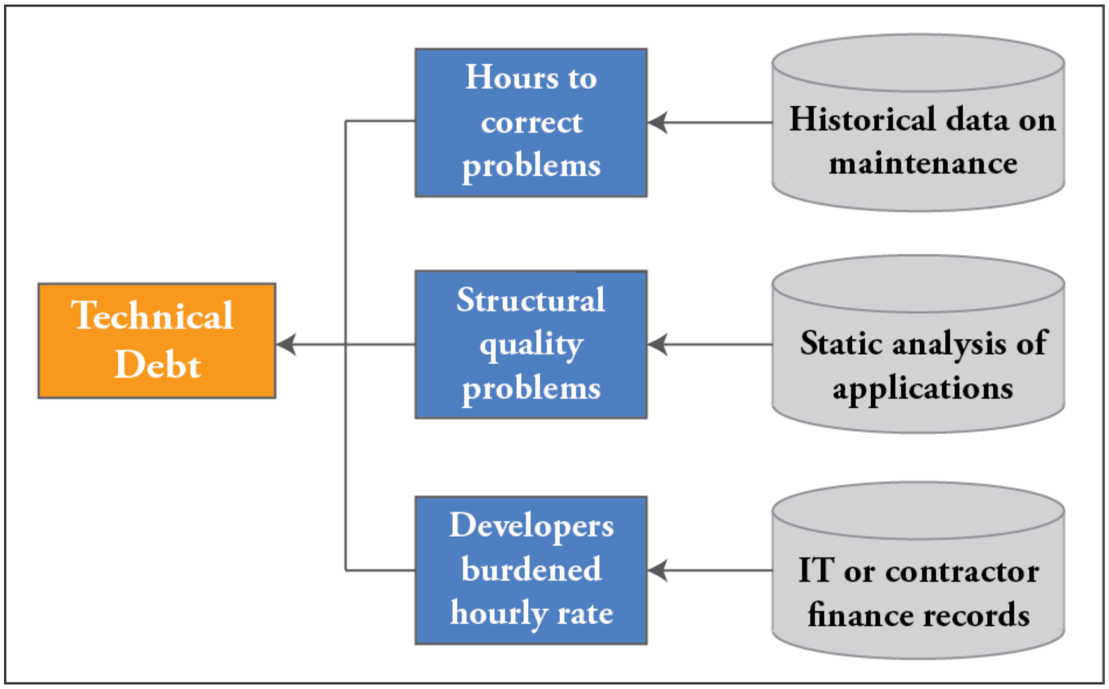
Hours to correct problems
Historical data on maintenance
Technical Debt
Structural quality problems
Static analysis of applications
Developers burdened hourly rate
IT or contractor finance records

Figure 3. Components and Data Sources to Calculate Technical Debt

Highlights

Calculating Technical Debt is a function of three variables—the number of must-fix problems in an application, the time required to fix each problem, and the cost for fixing a problem.

Number of must-fix problems: The number of structural problems in an application can be measured through the static analysis of the application's source code. However, with limited budgets, an IT organization will never fix all of the problems in an application. Therefore each of the structural problems detected through static analysis must be weighted by its potential severity. If severity scores are grouped into categories—for instance low, medium and high—then IT executives can determine what percentage of problems in each category are must-fix.

Time to correct problems: The time to fix a structural quality problem includes the time to analyze the problem, understand the code and determine a correction, evaluate potential side-effects, implement and test the correction, and release the correction into operations.

Cost to fix problems: This variable can be set to the average burdened rate for the developers assigned to fix structural problems. Although burdened hourly rates may vary by experience and location, we have found that a rate of between $70 and $80 per hour reflects the average costs for most IT organizations. If an organization's labor rates vary widely, this variable can also be measured as a frequency distribution of costs.

CAST has pioneered the measurement of Technical Debt using its Application Intelligence Platform (AIP), which analyzes the complete source code of an application across its constituent layers, languages, and technologies to enable a thorough evaluation of its structural quality. Violations of good architectural coding practice are detected and weighted by the severity of their impact.

The anonymous results from many of the applications analyzed by AIP have been collected into a repository called Appmarq, which at the end of 2011 comprised 745 applications containing 365 million lines of code (11.3 million backfired function points). The applications were submitted between 2008 and 2011 by 160 organizations located primarily in the United States, Europe, and India. Although this sample may not characterize the global population of IT business applications, it does emerge from the largest sample of applications ever analyzed for structural quality and therefore these results present a starting point for measuring and analyzing Technical Debt.

There is not a single method of measuring Technical Debt, and based on significant research with industry leaders CAST had devised a method for calculating Technical Debt figures in the Appmarq repository for our annual CRASH Report (CAST Report on Application Technical Health) 2011-2012. This calculation is shown in Figure 4.

Highlights

A typical application accrues $361,000 of Technical Debt for each 100,000 lines of code, and applications of 300,000 or more lines of code carry more than $1 million of Technical Debt ($1,083,000).

$$((.1\ \Sigma\ LSV + .25\ \Sigma\ MSV + .5\ \Sigma\ HSV) \times 1 \text{ hour} \times \$75)$$

Where, LSV = Low Severity Defects
MSV = Medium Severity Defects
HSV = High Severity Defects

Figure 4. CAST's Formula for Calculating Technical Debt

For our initial analyses we assumed that only 50% of high severity problems, 25% of moderate severity problems, and 10% of low severity problems will ultimately be corrected in the normal course of operating the application. Although defect fix times usually take longer, and in a few instances much longer, we initially chose a very conservative 1 hour per fix. Finally, it appears that a reasonable average cost per developer hour is in the region of $75 per hour when averaged across all the geographically dispersed sources of development activity utilized by most IT organizations. The resulting equation after setting parameters at these levels is a very conservative estimate of Technical Debt. In fact, we believe that in most cases the amount of Technical Debt will be much higher. In future analyses we may adjust the parameters of the equation in Figure 4 as data from industrial experience becomes more available. The important consideration is that Technical Debt can be estimated by an equation whose parameters can be adjusted by an organization to best reflect its own experience.

The Technical Debt formula presented only provides a basis for benchmarking. We encourage organizations to adjust the parameters in this formula to best fit their own maintenance and structural quality objectives, experiences, and costs. However, to benchmark against industry data, organizations that have customized these parameters will need to recalculate their Technical Debt according to the standard formula presented here.

V. Technical Debt Industry Data

We first calculated the Technical Debt individually for each of the 745 applications in the Appmarq sample and resulting analysis showed an average Technical Debt of $3.61 per line of code. Consequently, a typical application accrues $361,000 of Technical Debt for each 100,000 lines of code, and applications of 300,000 or more lines of code carry more than $1 million of Technical Debt ($1,083,000). These Technical Debt figures are a conservative estimate of the cost to remediate only the must-fix problems based on assuming only one hour per repair.

Although IT organizations could estimate their total Technical Debt by multiplying an estimate of the size of their application code base by $3.61, it would be more accurate to analyze it by technology and language type since significant differences were found. Technical Debt figures are reported in Table 1 for which there were at least 30 applications in the Appmarq repository, with the lowest figure being reported for ABAP and the highest for Java-EE. These figures may change year-on-year as the mix of application characteristics in each technology/language category change in our sample so we urge caution in interpreting these figures as industry benchmarks.

	Application Technology / Language					
	.NET	ABAP	C	C++	Java-EE	Oracle Forms
Count	63	72	44	30	474	45
Mean	$3,090	$425	$2,622	$4,326	$5,417	$4,565
Median	$2,267	$406	$2,183	$2,414	$5,133	$1,124
Maximum	$96	$58	$23	$21	$65	$485
Minimum	$13,524	$1,422	$12,822	$38,078	$49,720	$30,231
Standard Deviation	$2,703	$226	$2,584	$7,018	$3,913	$6,702

Table 1. Technical Debt per KLOC by Technology/Language Category

While the maximum and minimum values appear large for all languages except ABAP, the standard deviations indicate that the greatest variability in Technical Debt figures occurred for C++, Java-EE, and Oracle Forms. These results demonstrate that even for applications developed using the same language and technology, Technical Debt results can vary widely. Consequently, to be used effectively for management decisions, Technical Debt should be measured and analyzed individually for each application rather than using an average value across applications.

VI. Different Types of Technical Debt

Although Technical Debt is measured simply as violations of good structural quality, these violations consist of different types of threats and costs to the business. For management to make decisions on resource allocation to eliminate the violations, they need to distinguish the type of threat and prioritize the importance of Technical Debt in each area. To aid in this determination, CAST has identified five software quality

Highlights

70% of Technical Debt is contained in the Health Factors related to IT costs, Changeability and Transferability

characteristics called Health Factors that represent risks to the business or costs to IT as described in Table 2: Robustness, Performance, Security, Transferability, and Changeability.

Issue	Health Factor	Definition
Risks to the business	Robustness	Stability of an application and the likelihood of introducing new defects when modifying it
	Performance	The responsiveness of an application
	Security	An application's ability to prevent unauthorized intrusions
Costs to IT	Transferability	The ease with which a new team can understand the application and quickly become productive working on it
	Changeability	An application's ability to be easily and quickly modified

Table 2. Structural Quality Health Factors

Figure 5 shows the distribution of Technical Debt across the five application Health Factors in the Appmarq sample as detailed in the CRASH Report 2011-2012. Seventy percent of the Technical Debt was contained in the Health Factors related to IT costs, Changeability and Transferability. We cannot determine from the data the cause of this imbalance, whether IT organizations are spending more time eliminating Technical Debt related to business risk or whether Technical Debt is disproportionately created in IT cost-related factors. Nevertheless, a single high severity violation related to business risk can be devastating if it eventually causes an operational problem or failure.

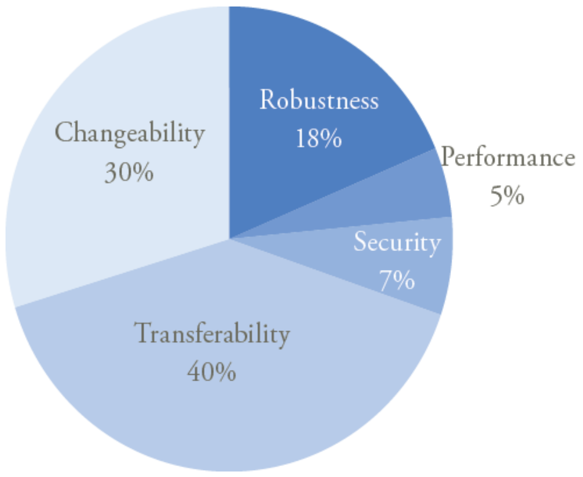

Figure 5. Technical Debt by Application Health Factor

Similar to the results calculating the total Technical Debt of an application, the results for the Technical Debt related to each Health Factor differ by technology/language category, as shown in Figure 6. Although the comparative percentages of Technical Debt remain generally consistent among Health Factors across technology/language categories, considerable variation is apparent. In particular, the Technical Debt scores for Robustness appear much higher for ABAP and Oracle Forms.

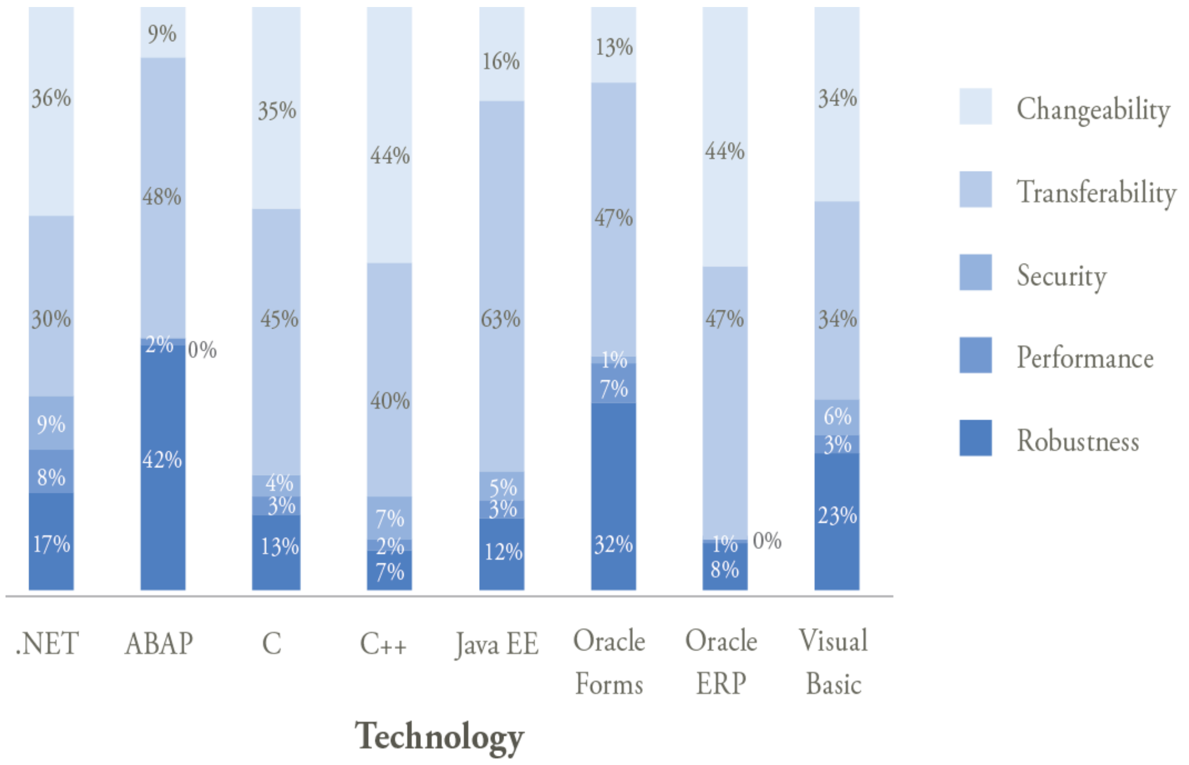

Figure 6. Technical Debt by Health Factor within Technology/Language Category

Trying to make decisions about retiring Technical Debt at a global level can be overwhelming and it is often difficult to visualize the expected payoff. However, the analysis and measurement of Technical Debt can guide critical management decisions about how to allocate resources for reducing business risk and IT cost when based on its constituent parts. Executives can set specific reduction targets based on strategic quality priorities with an expectation of the benefit to be achieved based on the definitions in Table 2. For instance, removing the highest severity violations of Robustness practices reduces the risk of catastrophic operational crashes, improving IT's ability to achieve availability targets. As IT collects more data, management will be able to develop a quantitative understanding of how much Technical Debt related to Robustness it can sustain in an application without risking its availability goals.

Highlights

The Technical Debt Management Cycle begins with executive business priorities that are translated to Technical Debt targets for each application.

VII. Managing Technical Debt

The Technical Debt Management Cycle is a 7-step process for analyzing and measuring Technical Debt, presented in Figure 7, that relies on relating it to IT and business priorities. This process begins with executive business priorities that are translated to Technical Debt targets for each application. It is vital that actions for achieving these targets are planned and progress is tracked at each release and periodically reported to the business.

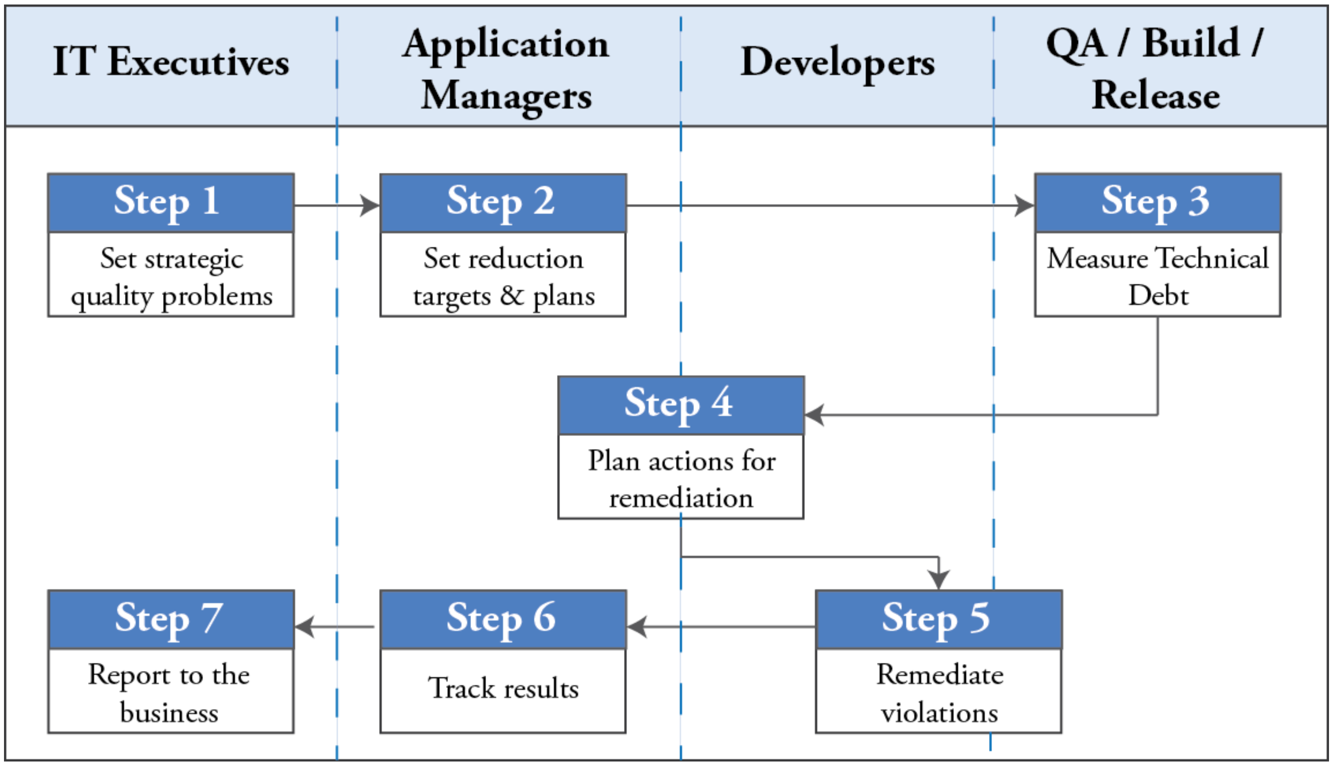

Figure 7. Flow of the Technical Debt Management Cycle

Highlights

Structural quality analysis and measurement provides the data needed to optimize the balance between business risk factors and internal IT cost factors.

Step 1: Set strategic quality priorities. Since application budgets and time are constrained, IT executives must determine the most critical structural quality objectives for each of the application services it offers to the business and provide unambiguous guidance to application managers. In many cases they will need to set these priorities in consultation with the application managers, since executives will generally lean toward business risk factors and structural quality priorities will typically vary for different types of applications. For instance, in highly regulated industries the most critical issue may be protecting customer data, which would make Security the highest priority. In highly competitive markets, business agility may be the most critical issue so Changeability would be a priority because of its impact on the speed of delivering new functionality. And on online retail application may need to satisfy several priorities, such as Robustness and Security.

Applications with large and growing levels of Technical Debt related to IT costs may need to prioritize these factors over business risk issues since the application architecture can degrade to the point that it becomes unable to serve its business mission. Thus, IT executives must determine the balance between business risk factors and their own internal IT cost factors. Fortunately, structural quality analysis and measurement provides the data needed to optimize the balance among strategic priorities.

Step 2: Establish reduction targets and plans. Application managers will need to translate the structural quality priorities into specific targets for their application. These targets could be stated as thresholds such as ‘no high severity Performance defects in the operational code base’, or ‘sustain Transferability scores at or above 2.8 across releases’. While targets can be set for all five Health Factors, the strategic priorities passed down by executive management provide the guidance needed to allocate resources toward the most critical targets.

In every release there is tension between quickly providing new functionality to the business versus reducing Technical Debt in priority areas, and as a result structural quality targets can rarely be achieved in the span of one or two releases. Rather, Technical Debt reduction and management needs to be treated as a standard component of maintenance or sprint planning. Application managers must allocate time for structural quality improvement into their application maintenance plans, or in an agile environment allocate a percentage of stories in each sprint to remediating structural quality problems. Since developers frequently participate in these planning activities they should be aware of the strategic structural quality priorities against which they are expected to plan.

Highlights

Serious architectural flaws that span layers of the application and may be written in different languages are often the most devastating during operations and have been shown to be the most time-consuming to fix.

Step 3: Measure Technical Debt. The structural quality flaws underlying Technical Debt can be measured at numerous points in the software life cycle, such as freeware static analysis tools used by developers to measure Technical Debt as part of their unit testing regimen before submitting code to a build. However, analysis at this level cannot detect serious architectural flaws that span layers of the application and may be written in different languages. These multi-component flaws are often the most devastating during operations and have been shown to be the most time-consuming to fix[5]. Consequently the most effective time to measure structural quality is after a build or before release when the entire application can be analyzed as a whole.

Application structural quality analysis is usually performed by the build team or quality assurance, or in an Application Intelligence (AI) Center dedicated to this function. The results of these analyses should be fed back to the development team as summary measures for each Health Factor and a list of violations on which these measures are based. These analyses should be retained as a baseline to assess progress toward strategic structural quality priorities.

Step 4: Plan remediation actions. At the beginning of each release planning cycle or sprint, the application manager and development team can use the structural quality analysis from the previous release to prioritize a list of violations for remediation that will help to achieve the application's structural quality targets. The team should select as many high-priority violations to remediate as can be addressed by the resources allocated during the cycle or sprint to reduce Technical Debt. Thresholds should be set for various Health Factors that must be sustained at release, and these thresholds can be used to ensure Technical Debt does not increase as a result of development activities that are separate from the remediation of Technical Debt.

To maintain the continuity of progress in reducing Technical Debt, the list of prioritized violations should be revised and updated after each release. The rate of progress in remediating the backlog of violations can be compared to executive priorities and timelines to determine if the resource allocation needs to be adjusted in future cycles or sprints, ensuring Technical Debt planning continues throughout the application lifespan as part of the longer strategic plan.

Step 5: Remediate violations. The development team addresses the violations tagged for remediation during the cycle or sprint as part of their normal development or maintenance activities. These remediation activities may involve refactoring poorly designed code or correcting specific coding weaknesses. Testing and static analysis should be used to verify

Highlights

Technical Debt data can be used as input for estimating and tracking the long term cost of ownership for the application.

that the flaw has been successfully remediated. The team should keep track of the time required to remediate structural flaws to help with more accurate estimates on the number of remediation tasks that can be undertaken in future cycles or sprints.

If a specific type of flaw has been detected numerous times across the code base, the development team should, as time allows, identify the root cause of the flaw and take steps to eliminate it. This is an appropriate topic for retrospectives at the end of agile sprints. Results of these root cause analyses should be reported back to central team conducting the structural quality analyses so they can detect trends across the organization.

Step 6: Track results. The application manager should track the ongoing results of Technical Debt remediation efforts, compare the results against established targets, and share them with the development team. Significant deviations against expected progress should be addressed with the development team. When planning upcoming cycles or sprints, progress in retiring Technical Debt should be reviewed to develop achievable commitments based on the resources allocated.

Periodically the application manager and IT executives should review progress in retiring Technical Debt and reaffirm application targets against the strategic structural quality priorities, adjusting the targets if strategic priorities have changed. Technical Debt data can be used as input for estimating and tracking the long term cost of ownership for the application. Structural quality data needed for upcoming management decisions regarding the application should be discussed.

Step 7: Report to the business. Following review with the application managers, IT executives should report the status of Technical Debt to the business. With the direct relation between Health Factors and critical aspects of either business risk or IT cost, Technical Debt can be translated into terms easily understood by the business like reductions in risks such as security breaches or outages, increased capability for business agility, or reductions in IT's long term costs. This information can also be reported to corporate auditors responsible for assessing risks to the business.

However, translating Technical Debt into business risks requires the business to articulate its costs and the losses experienced when applications suffer outages, performance degradation, security breaches, data corruption, and similar events. The discussion of Technical Debt and its potential liabilities initiates a dialogue that will provides business with greater understanding of the IT risks and costs and how they relate to business issues. Consequently the Technical Debt Management Cycle provides a vehicle for greater alignment between the business and IT.

VIII. Case Study 1

A large insurer had a claims management system with 700,000 lines of code that managed 8 million claims per year from a client base of 10 million customers. Development of new functionality suffered as a result of substantial Technical Debt, and was typically accompanied by high defect rates.

To gain control over Technical Debt and reduce maintenance costs, the executive in charge of this application mandated that future development would be subjected to structural quality measurement and baseline targets were set for maintainability. Development teams were held accountable for making steady progress toward the maintainability targets release by release, and were presented the results of each structural quality analysis.

Over the course of four years, they saw the following results:

- Maintainability targets were stabilized against the baseline target even though the application code base grew by 40%.
- Defect rates in system test and operations fell 56% as Technical Debt was remediated and the application became easier to understand and maintain, as shown in Figure 8.
- Delivery time was reduced by 60% as a result of more maintainable code and less interference from rework.
- Maintenance costs for this application were reduced by 20% within the first three years.

Highlights

The claims management system for a larger insurer saw defect rates in system test and operations fall 56% as Technical Debt was remediated and the application became easier to understand and maintain.

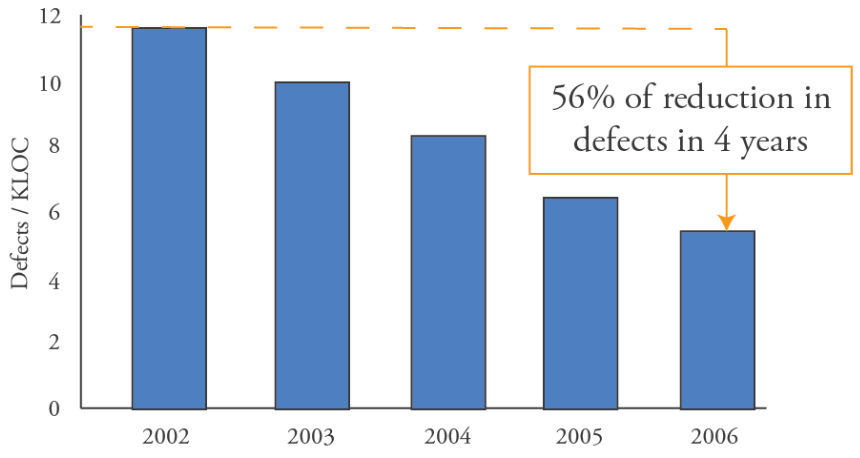

Figure 8. Reduction in Defect Rates for a Large Insurance Application

IX. Case Study 2

CAST performed structural analysis of Technical Debt on the billing system of a large telecommunications company. Management's objectives were to reduce rework and outages, as well as gain control over outsourced maintenance costs. If managing Technical Debt addressed the objectives, then the process would be deployed to reduce Technical Debt in other critical applications.

Highlights

For the billing system of a large telecom company , reducing rework by lowering the number of defects resulted in an estimated cost savings of $2.7 million in the first year.

Technical Debt was measured at an initial release and critical violations of good architectural and coding practice (anti-patterns) were targeted for remediation. Over the subsequent three releases, the anti-patterns constituting Technical Debt were steadily reduced, as shown in Figure 9. Along with this reduction in Technical Debt was a highly correlated reduction in system test and operational defects. A decline in operational problems was also observed, which was correlated with reductions in defect rates, although the data were not available to prove a causal relationship.

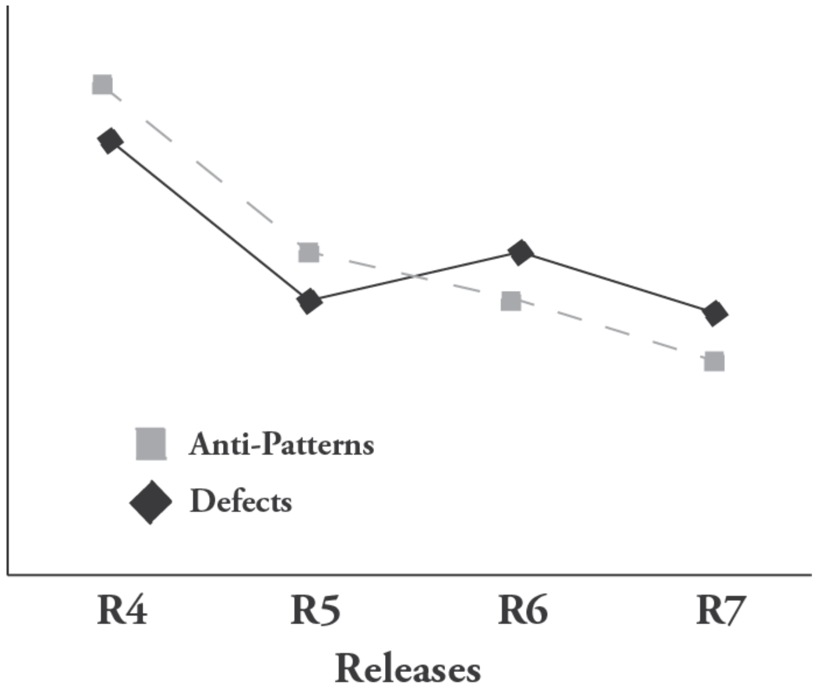

Figure 9. Reduction in Technical Debt and Defects across Releases

Based on the successful pilot, structural analysis was deployed to a portfolio of critical applications to reduce and control Technical Debt. Figure 10 shows structural quality analysis was initiated on this particular application at release 8. The effort to remediate Technical Debt is correlated with a reduction in system test and operational defects over the next nine releases until it appears to stabilize at release 11.2. The same pattern was observed with other applications.

Reducing rework by lowering the number of defects resulted in a substantial reduction in maintenance costs. IT estimated cost savings of $2.7 million in the first year on this particular application as a result of rework and outage reductions as well as better management of outsourced services.

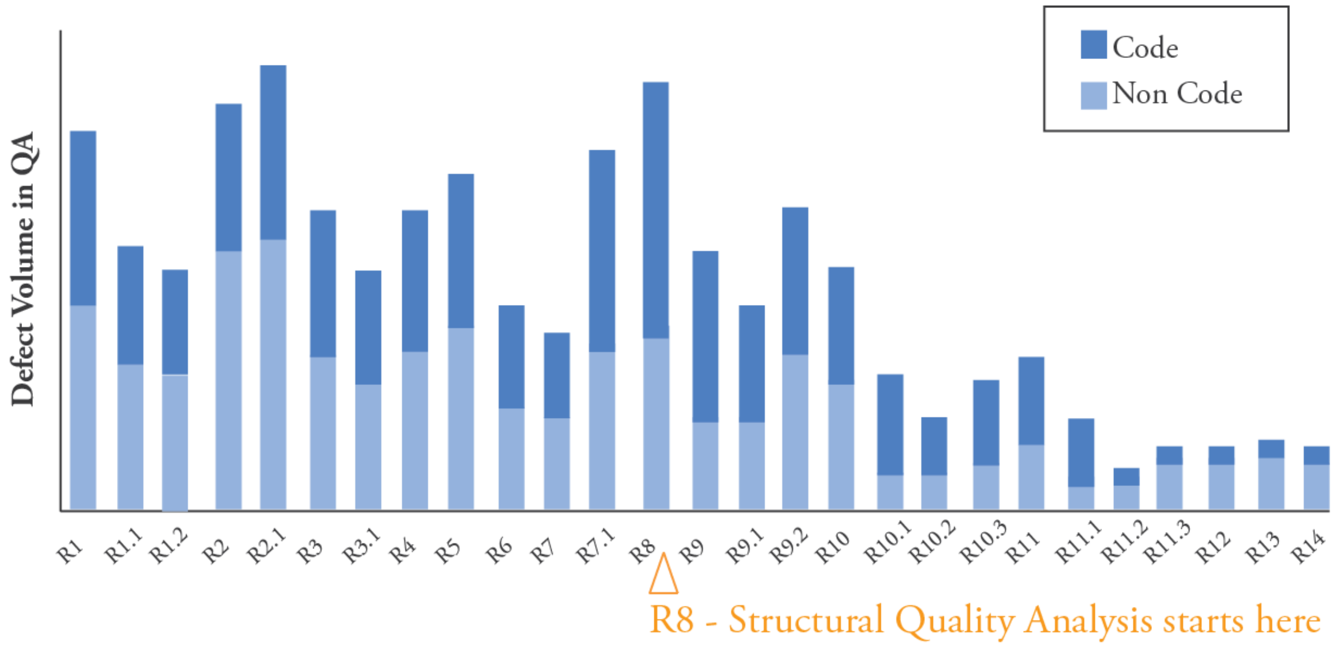

Figure 10. Defect Volume Reduction across Releases

Highlights

Fear of and resistance to the collection and use of structural quality data can be substantially reduced when it is perceived as a tool for continual improvement rather than an adjunct to performance appraisals.

X. Integrating Technical Debt with Lean IT

Many IT organizations are exploring how the lean practices popularized by the Toyota Production System will work in their environment. One of the pillars of Toyota's House of Quality is Jidoka, the automated assistance to eliminating quality problems. The Technical Debt Management Cycle fits well with the practice of Jidoka as IT is using automated structural analysis to eliminate quality problems in its applications. The two case studies presented above demonstrate that managing Technical Debt can successfully reduce waste and costs in application development and maintenance. Consequently, the management of Technical Debt is central to creating a lean IT operation.

Central to the lean philosophy is to detect quality problems as early as possible and eliminate their causes. The data from structural quality analysis and measurement can be fed forward in the development process to assist both of these objectives. If development teams are conducting architecture, design, or code inspections, information about the most frequent types of structural quality flaws can be added to inspection checklists to detect and eliminate these problems before components are submitted to a build. An additional benefit of inspections is the substantial learning, especially about architectural issues, that occurs while conducting them.

Another way to eliminate a cause of structural quality problems is to incorporate information about frequent violations into training to ensure developers have the knowledge and skills required to avoid them. The data from structural quality analysis and measurement activities provides a rich source of information about developer capabilities that can be addressed through training and mentoring. Fear of and resistance to the collection and use of structural quality data can be substantially reduced when it is perceived as a tool for continual improvement rather than an adjunct to performance appraisals.

The various practices in the Technical Debt Management Cycle can be expanded and adjusted to a wide range of IT functions. For instance, portions of the application development and maintenance work in most large IT organizations have been outsourced to external vendors. Even with much of the work occurring outside the organization, Steps 1, 2, 3, 5, and 6 are still relevant to ensuring that the structural quality of outsourced work does not present unwarranted risks and costs to IT and the business. Structural quality targets could be written into contracts and compliance could be tracked through application-level structural analysis.

Highlights

Practitioners of agile methods need to adapt the Technical Debt Management Cycle to their short delivery cycles if they are to avoid dramatic increases in an application's lifetime costs.

The essence of lean operations is to eliminate waste. The largest source of waste in most IT organizations is the time and resources spent fixing both functional and structural defects. The Technical Debt Management Cycle provides practices that directly reduce the waste caused by structural quality problems and are a critical component of any program to implement lean IT.

XI. Applying Technical Debt to Agile Methods

The agile methods community has embraced lean principles and is adapting them to short delivery cycles. Technical Debt has been an especially acute issue within the agile methods community because of their short cycle times. Agile teams frequently implement code with known structural weaknesses because it allows them to release production-ready code within the short, demanding schedules of agile methods. There is rarely enough time to refactor or correct all the structural problems that should be addressed, so the effort and cost of remediating structural weaknesses is shifted from development into maintenance.

To repay Technical Debt, agile teams incorporate refactoring the most egregious structural flaws into their backlog of stories to implement in future iterations. Unfortunately, the business usually prioritizes new functionality over refactoring structural flaws, allowing few if any refactoring stories to be removed from the backlog. As a result, Technical Debt grows unrestrained over an application's life.

Practitioners of agile methods need to adapt the Technical Debt Management Cycle to their short delivery cycles if they are to avoid dramatic increases in an application's lifetime costs. Although some in the agile community have argued that Technical Debt is not measureable, the case studies presented above have demonstrated that it can be measured and used for controlling maintenance results. The use of techniques such as structural analysis is critical for producing the measures needed to protect allocated time in each sprint or cycle to refactoring Technical Debt.

XII. Conclusion

This white paper defines Technical Debt in a form that can be addressed strategically by IT executives, managed by application managers, measured by quality assurance or release management, and remediated by developers. More importantly, the ability to analyze Technical Debt into its constituent quality Health Factors allows it to be directly related to critical business issues such as availability, agility, cost, and privacy protection—and subsequently can be used to reduce business risk and IT cost.

XIII. References

1. Cunningham, W. (1992). The WyCash portfolio management system. Proceedings of OOPSLA'92. Alameda, CA: IEEE Computer Society Press.

2. Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68 (9), 1060–1076.

3. McConnell, S. (2007). Technical Debt. http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

4. Jones, C. & Bonsignour, O. (2011). The Economics of Software Quality. Upper Saddle River, NJ: Prentice Hall.

5. Li, Z., Madhavji, N.H., Murtaza, S.S., Gittens, M., Miranskyy, A.V., Dodwin, D., & Cialini, E. (2011). Characteristics of multi-component defects and architectural hotspots: A large system case study. Empirical Software Engineering, 16 (5), 667-702.

About the Author

Dr. Bill Curtis
Senior Vice President and Chief Scientist

Bill Curtis is an industry luminary who is responsible for influencing CAST's scientific and strategic direction, as well as helping CAST educate the IT market to the importance of managing and measuring the quality of its software. He is best known for leading the development of the Capability Maturity Model (CMM) which has become the global standard for evaluating the capability of software development organizations.

Prior to joining CAST, Dr. Curtis was a Co-Founder of TeraQuest, the global leader in CMM-based services, which was acquired by Borland. Prior to TeraQuest, he directed the Software Process Program at the Software Engineering Institute (SEI) at Carnegie Mellon University. Prior to the SEI he directed research on intelligent user interface technology and the software design process at MCC, the fifth generation computer research consortium in Austin, Texas. Before MCC he developed a software productivity and quality measurement system for ITT, managed research on software practices and metrics at GE Space Division, and taught statistics at the University of Washington.

Dr. Curtis holds a Ph.D. from Texas Christian University, an M.A. from the University of Texas, and a B.A. from Eckerd College. He was recently elected a Fellow of the Institute of Electrical and Electronics Engineers for his contributions to software process improvement and measurement. In his free time Dr. Curtis enjoys traveling, writing, photography, helping with his daughter's homework, and University of Texas football.

About CAST

CAST is a pioneer and world leader in Software Analysis and Measurement, with unique technology resulting from more than $90 million in R&D investment. CAST provides IT and business executives with precise analytics and automated software measurement to transform application development into a management discipline. More than 650 companies across all industry sectors and geographies rely on CAST to prevent business disruption while reducing hard IT costs. CAST is an integral part of software delivery and maintenance at the world's leading IT service providers such as IBM and Capgemini.

Founded in 1990, CAST is listed in NYSE-Euronext (Euronext: CAS) and services IT intensive enterprises worldwide with a network of offices in North America, Europe, and India.

North America
373 Park Avenue South
New York, NY 10016
Phone: +1 212-871-3330

Europe
3, rue Marcel Allegot
92190 Meudon - France
Phone: +33 1 46 90 21 00