The CRASH Report - 2011/12

(CAST Report on Application Software Health)

Summary of Key Findings

CAST

Achieve Insight. Deliver Excellence.

CAST
ACHIEVE INSIGHT. DELIVER EXCELLENCE.

Contents

CAST
Achieve Insight. Deliver Excellence.

Introduction

365 million lines of code

745 applications

160 organizations

Overview

This is the second annual report produced by CAST on global trends in the structural quality of business applications software. These reports highlight trends in five structural quality characteristics—Robustness, Security, Performance, Transferability, and Changeability—across technology domains and industry segments. Structural quality refers to the engineering soundness of the architecture and coding of an application rather than to the correctness with which it implements the customer's functional requirements. Evaluating an application for structural quality defects is critical since they are difficult to detect through standard testing, and are the defects most likely to cause operational problems such as outages, performance degradation, breaches by unauthorized users, or data corruption.

This summary report provides an objective, empirical foundation for discussing the structural quality of software applications throughout industry and government. It highlights some key findings from a complete report that will provide deeper analysis of the structural quality characteristics and their trends across industry segments and technologies. The full report will also present the most frequent violations of good architectural and coding practice in each technology domain. You can request details on the full report at:
http://research.castsoftware.com.

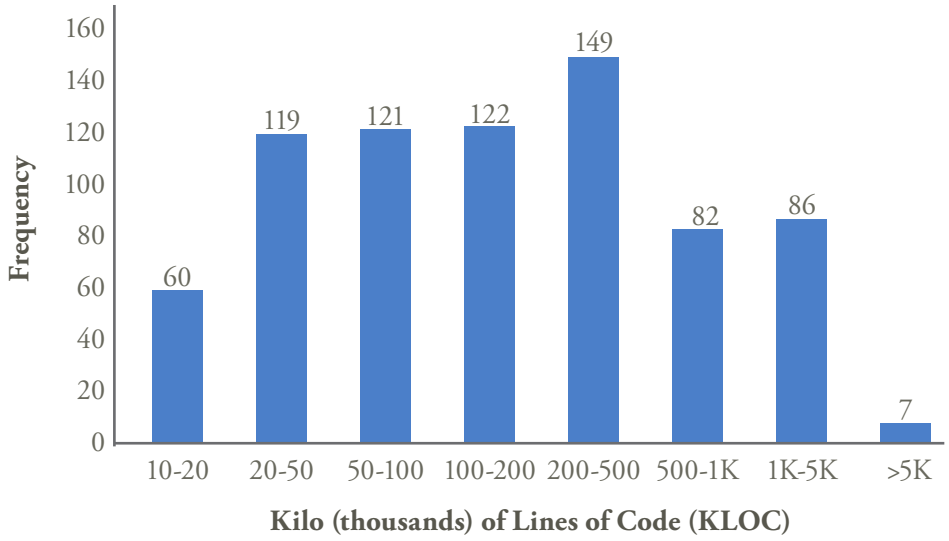
The Sample

The data in this report are drawn from the Appmarq benchmarking repository maintained by CAST, comprised of 745 applications submitted by 160 organizations for the analysis and measurement of their structural quality characteristics, representing 365 MLOC (million lines of code) or 11.3 million Backfired Function Points. These organizations are located primarily in the United States, Europe, and India. This data set is almost triple the size of last year's sample of 288 applications from 75 organizations comprising 108 MLOC.

The sample is widely distributed across size categories and appears representative of the types of applications in business use. Figure 1 displays the distribution of these applications over eight size categories measured in lines of code. The applications range from 10 KLOC (kilo or thousand lines of code) to just over 11 MLOC. This distribution includes 24% less than 50 KLOC, 33% between 50 KLOC and 200 KLOC, 31% between 201 KLOC and 1 MLOC, and 12% over 1 MLOC.

As is evident in Table 1, almost half of the sample (46%) consists of Java-EE applications, while .NET, ABAP, COBOL, and Oracle Forms each constituted between 7% and 11% of the sample. Applications with a significant mix of two or more technologies constituted 16% of the sample.

Figure 1. Distribution of Applications by Size Categories



As shown in Table 1, there are 10 industry segments represented in the 160 organizations that submitted applications to the Appmarq repository. Some trends that can be observed in these data include the heaviest concentration of ABAP applications in manufacturing and IT consulting, while COBOL applications were concentrated most heavily in financial services and insurance. Java-EE applications accounted for one-third to one-half of the applications in each industry segment.

Table 1. Applications Grouped by Technology and Industry Segments

Industry	.NET	ABAP	C	C++	Cobol	Java-EE	Mixed Tech	Oracle Forms	Oracle CRM/ERP	Other	Visual Basic	Total
Energy&Utilities	3	5	0	0	0	26	3	0	1	2	0	40
FinancialServices	5	0	0	2	39	46	50	3	0	4	1	150
Insurance	10	0	1	1	21	27	5	1	2	0	2	70
IT Consulting	11	11	2	2	13	51	6	0	6	1	6	109
Manufacturing	8	19	3	2	4	46	7	0	2	1	2	94
Other	3	2	1	2	1	11	9	1	0	0	0	30
Government	0	9	1	0	0	25	7	34	0	0	2	78
Retail	5	5	2	0	2	11	5	0	1	1	0	32
Technology	4	1	0	0	0	14	1	0	0	1	0	21
Telecom	2	7	4	0	0	82	24	0	0	1	1	121
Total	51	59	14	9	80	339	117	39	12	11	14	745

This sample differs in important characteristics from last year's sample, including a higher proportion of large applications and a higher proportion of Java-EE. Consequently, it will not be possible to establish year-on-year trends by comparing this year's findings to those reported last year. As the number and diversity of applications in the Appmarq repository grows and their relative proportions stabilize, we anticipate reporting year-on-year trends in future reports.

Terminology

LOC: Lines of code. The size of an application is frequently reported in KLOC (kilo or thousand lines of code) or MLOC (million lines of code).

Structural Quality: The non-functional quality of a software application that indicates how well the code is written from an engineering perspective. It is sometimes referred to as technical quality or internal quality, and represents the extent to which the application is free from violations of good architectural or coding practices.

Structural Quality Characteristics: This report concentrates on the five structural quality characteristics defined below. The scores are computed on a scale of 1 (high risk) to 4 (low risk) by analyzing the application for violations against a set of good coding and architectural practices, and using an algorithm that weights the severity of each violation and its relevance to each individual quality characteristic.

The quality characteristics are attributes that affect:
Robustness: The stability of an application and the likelihood of introducing defects when modifying it.
Performance: The efficiency of the software layer of the application.
Security: An application's ability to prevent unauthorized intrusions.
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.

We also measure:
Total Quality Index: A composite score computed from the five quality characteristics listed above.

Technical Debt: Technical Debt represents the effort required to fix violations of good architectural and coding practices that remain in the code when an application is released. Technical Debt is calculated only on violations that the organization intends to remediate. Like financial debt, technical debt incurs interest in the form of extra costs accruing for a violation until it is remediated, such as the effort required to modify the code or inefficient use of hardware or network resources.

Violations: A structure in the source code that is inconsistent with good architectural or coding practices and has proven to cause problems that affect either the cost or risk profile of an application.

PART I: Adding to Last Year's Insights

The distribution of security scores suggests some industry segments pay more attention to security

Finding 1—COBOL Applications Show Higher Security Scores

The distribution of Security scores across the current Appmarq sample is presented in Figure 2. The bi-modal distribution of Security scores indicates that applications can be grouped into two distinct types: one group that has very high scores and a second group with moderate scores and a long tail toward poor scores. The distribution of Security scores is wider than for any of the other quality characteristics, suggesting strong variations in the attention paid to security among different types of applications or industry segments.

Further analysis on the data presented in Figure 3 revealed that applications with higher Security scores continue to be predominantly large COBOL applications in the financial services and insurance sectors where high security for confidential financial information is mandated. These scores should not be surprising since COBOL applications run in mainframe environments where they are not as exposed to the security challenges of the internet. In addition, these are typically the oldest applications in our sample and have likely undergone more extensive remediation for security vulnerabilities over time.

The lower Security scores for other types of applications are surprising. In particular, .NET applications received some of the lowest Security scores. These data suggest that attention to security may be focused primarily on applications governed by regulatory compliance or protection of financial data, while less attention is paid to security in other types of applications.

Figure 2. Distribution of Security Scores

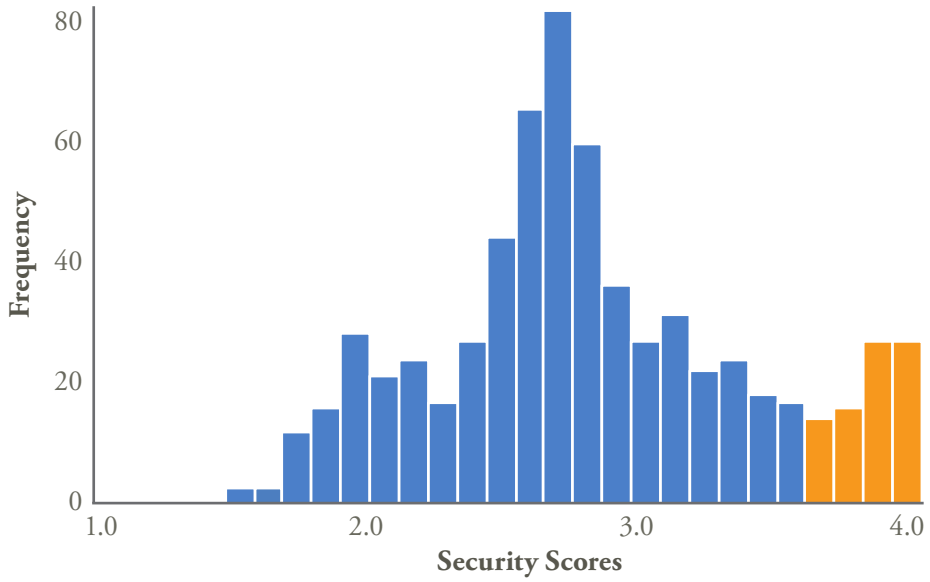


Figure 3. Security Scores by Technology

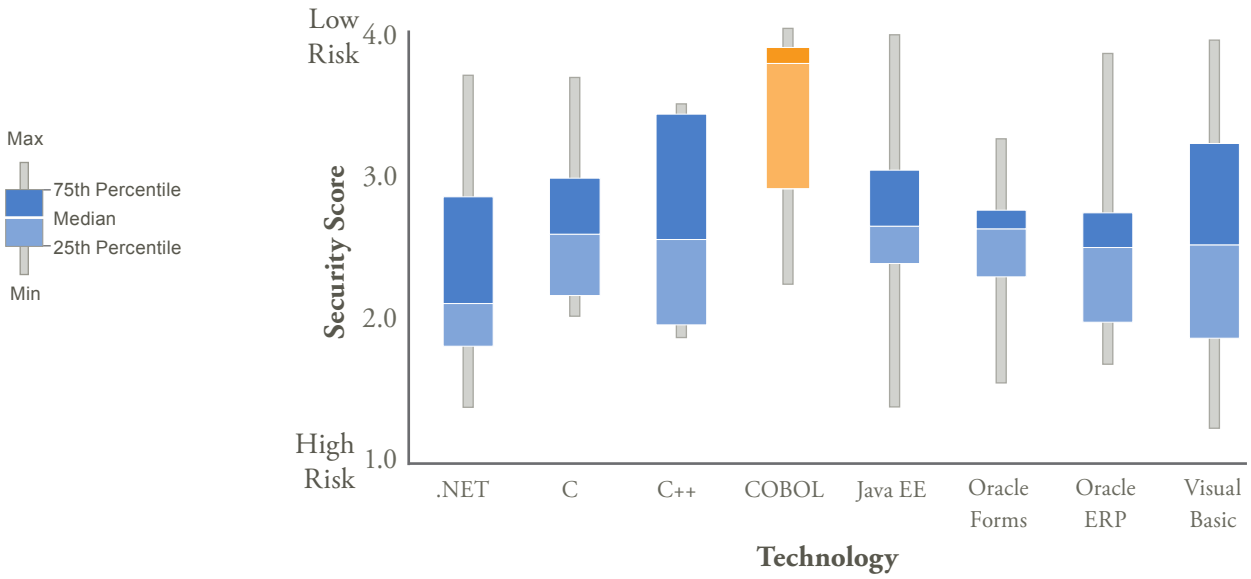
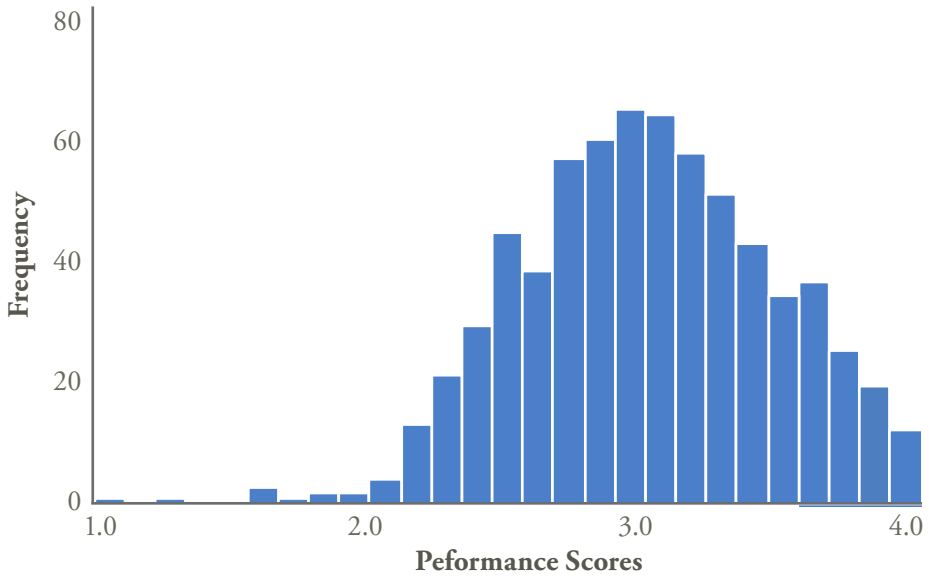


Figure 4. Distribution of Performance Scores

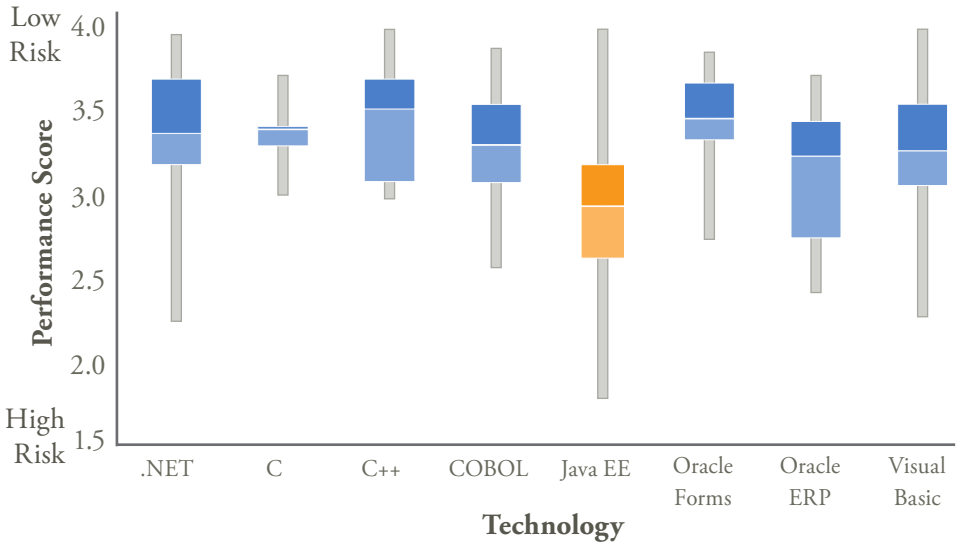


Finding 2—Performance Scores Lower in Java-EE

As displayed in Figure 4, Performance scores were widely distributed, and in general are skewed with the highest concentration towards better performance. These data were produced through software analysis and do not constitute a dynamic analysis of an application's behavior or actual performance in use. These scores reflect detection of violations of good architectural or coding practices that may have performance implications in operation, such as the existence of expensive calls in loops that operate on large data tables.

Further analysis of the data presented in Figure 5 revealed that Java-EE applications received significantly lower Performance scores than other languages. Modern development languages such as Java-EE are generally more flexible and allow developers to create dynamic constructs that can be riskier in operation. This flexibility is an advantage that has encouraged their adoption, but can also be a drawback that results in less predictable system behavior. In addition, developers who have mastered Java-EE may still have misunderstandings about how it interacts with other technologies or frameworks in the application such as Hibernate or Struts. Generally, low scores on a quality characteristic often reflect not merely the coding within a technology, but also the subtleties of how language constructs interact with other technology frameworks in the application and therefore violate good architectural and coding practices.

Figure 5. Performance Scores by Technology



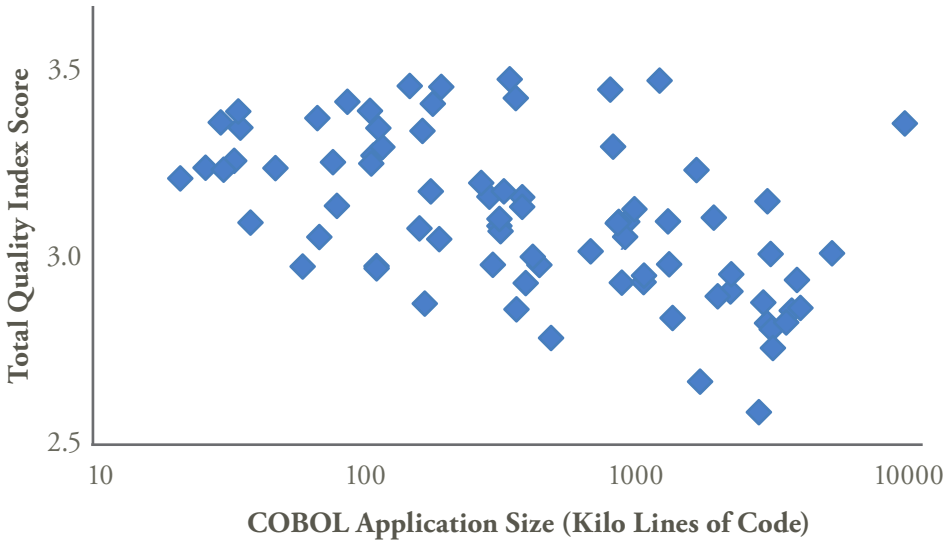
A negative correlation between size and quality is evident for COBOL applications

Finding 3—Modularity Tempers the Effect of Size on Quality

Appmarq data contradicts the common belief that the quality of an application necessarily degrades as it grows larger. Across the full Appmarq sample, the Total Quality Index (a composite of the five quality characteristic scores) failed to correlate significantly with the size of applications. However, after breaking the sample into technology segments, we found that the Total Quality Index did correlate negatively with the size of COBOL applications as is evident in Figure 6, where the data are plotted on a logarithmic scale to improve the visibility of the correlation. The negative correlation indicates that variations in the size of COBOL applications accounts for 11% of the variation in the Total Quality Index (R^2 = .11).

One explanation for the negative correlation between size and quality in COBOL applications is that COBOL was designed long before the strong focus on modularity in software design. Consequently, COBOL applications are constructed with many large and complex components. More recent languages encourage modularity and other techniques that control the amount of complexity added as applications grow larger. For instance, Figure 7 reveals that the percentage of highly complex components (components with high Cyclomatic Complexity and strong coupling to other components) in COBOL applications is much higher than in other languages, while this percentage is lower for the newer object-oriented technologies like Java-EE and .NET, consistent with object-oriented principles. However, high levels of modularity may present a partial explanation of the lower Performance scores in Java-EE applications discussed in Finding 2, as modularity could adversely impact the application's performance.

Figure 6. Correlation of Total Quality Index with Size of COBOL Applications



The increased complexity of components in COBOL is consistent with their much greater size compared with components in other languages. Figure 8 displays the aver-

Figure 7. Percentage of Components that are Highly Complex in Applications by Technology

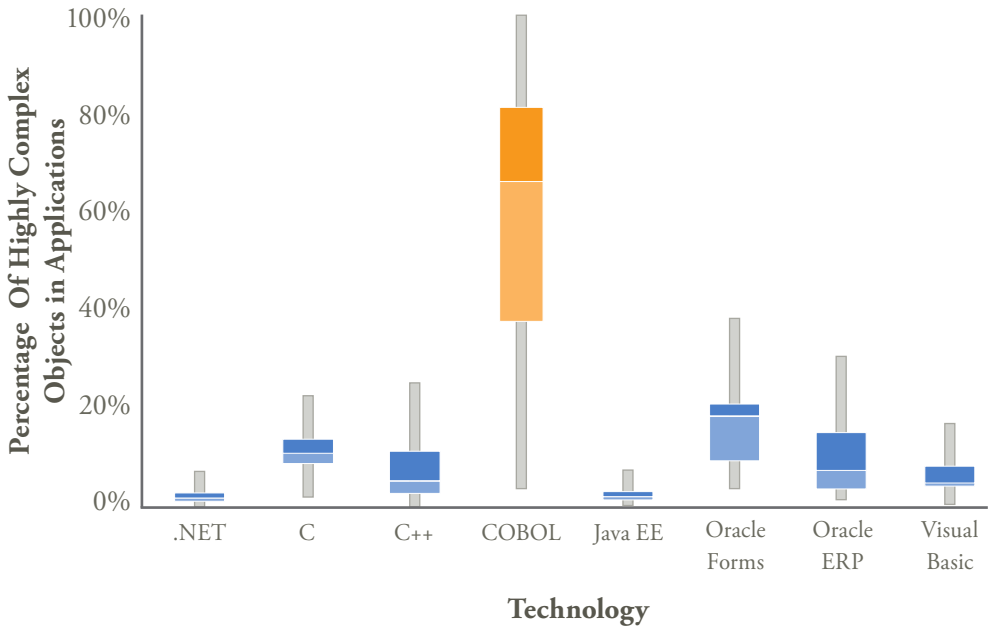
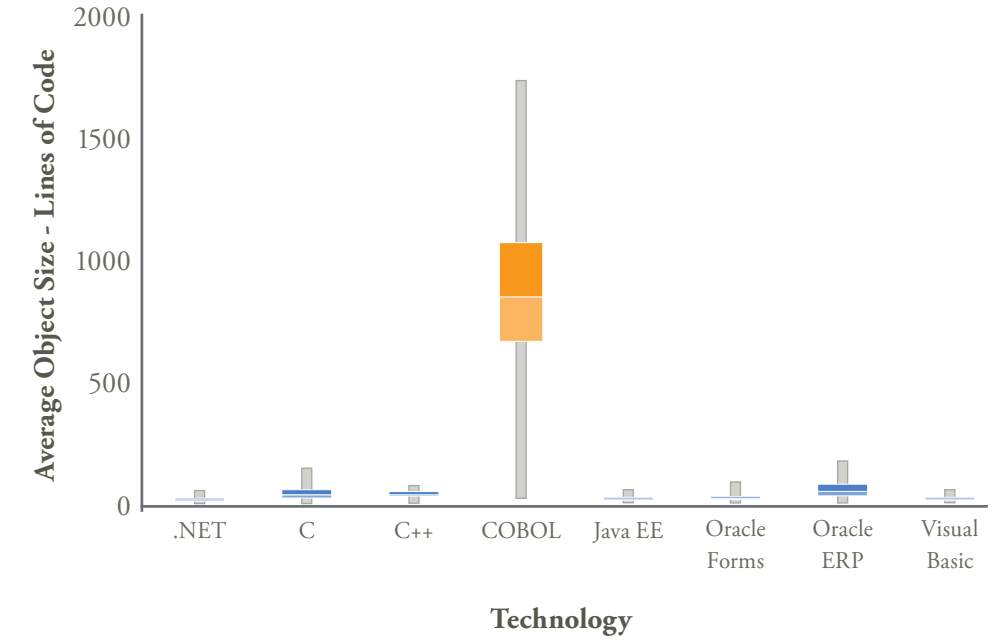


Figure 8. Average Object Size Comparison Across Different Technologies



age number of components per KLOC for applications developed in each of the technologies. While the average component size for most development technologies in the Appmarq repository is between 20 to 50 LOC, the average COBOL component is usually well over 600 LOC.

Measurements and observations of COBOL applications in the Appmarq repository suggest that they are structurally different from components developed in other technologies, both in size and complexity. Consequently we do not believe that COBOL applications should be directly benchmarked against other technologies because comparisons may be misleading and mask important findings related to comparisons among other, more similar technologies. Although we will continue reporting COBOL with other technologies in this report, we will identify any analyses where COBOL applications skew the results.

Finding 4—Maintainability Lowest in Government Applications

Transferability and Changeability are critical components of an application's cost of ownership, and scores for these quality characteristics in the Appmarq sample are presented in Figures 9 and 10. The spread of these distributions suggest different costs of ownership for different segments of this sample.

When Transferability and Changeability scores were compared by industry segment, the results presented in Figure 11 for Transferability revealed that scores for government applications were lower than those for other segments. The results for Changeability were similar, although the differences between government and other industry segments were not as pronounced. This sample includes government applications from both the United States and European Union.

Figure 9. Distribution of Transferability Scores

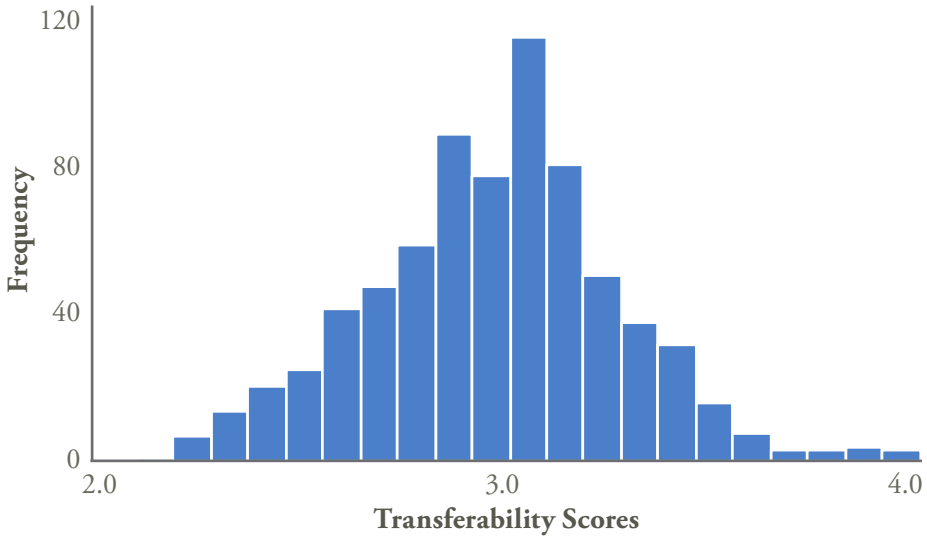


Figure 10. Distribution of Changeability Scores

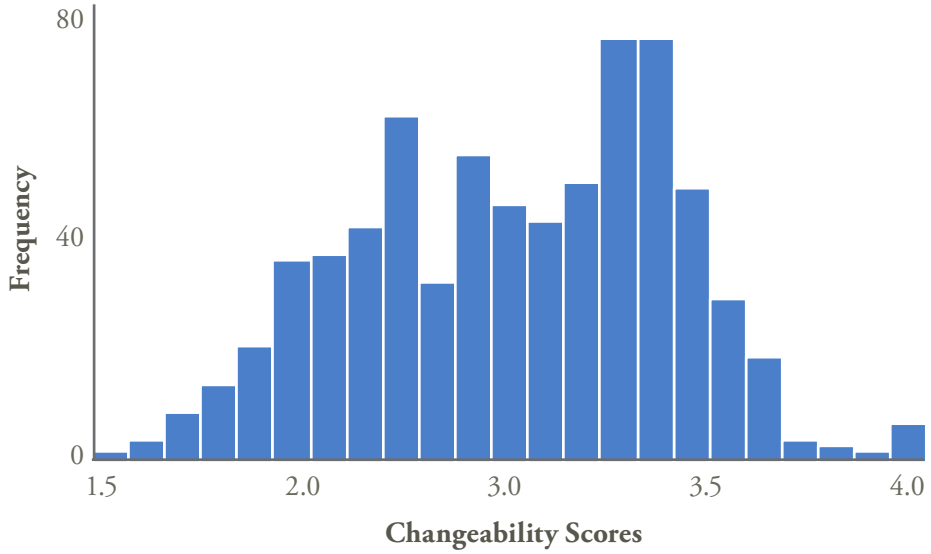
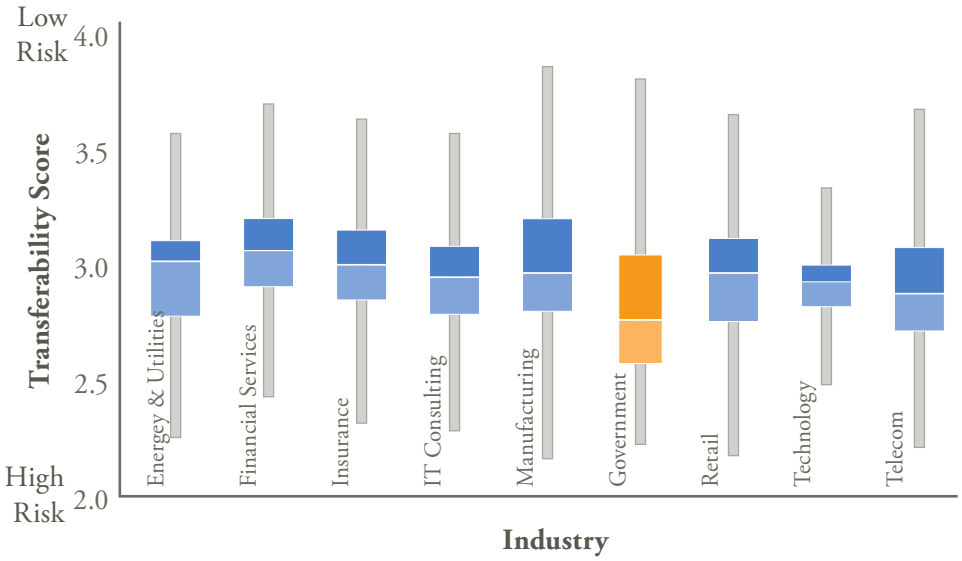


Figure 11. Transferability Scores by Industry Segment

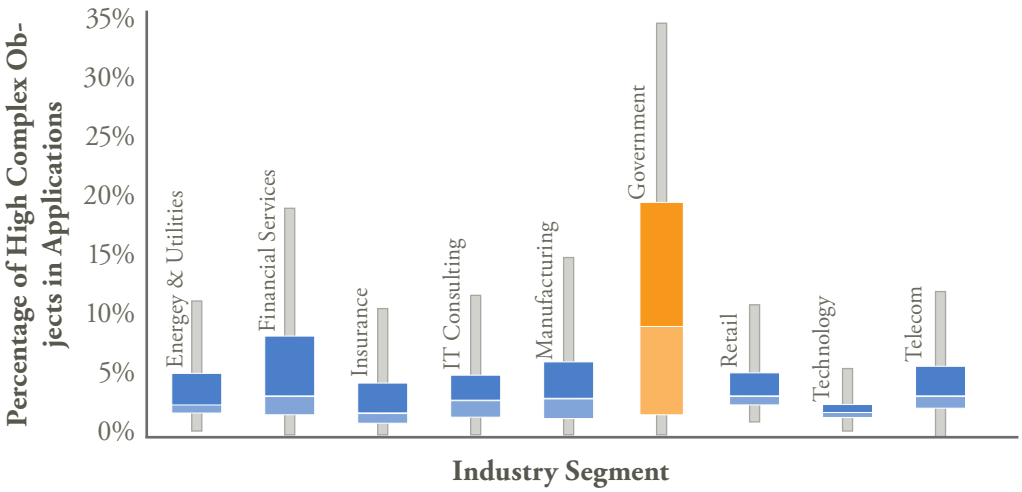


Government applications have the most chronic complexity profiles

Although we do not have cost data, these results suggest that government agencies are spending significantly more of their IT budgets on maintaining existing applications than on creating new functionality. Not surprisingly, the Gartner 2011 IT Staffing & Spending report stated that the government sector spends about 73% of its budget on maintenance, higher than any other segment.

The lower Transferability and Changeability scores for government agencies may partially result from unique application acquisition conditions. In the Appmarq sample, 75% of government applications were acquired through contracted work, compared to 50% of the applications in the private sector being obtained through outsourcing. Multiple contractors working on the same application over time, disincentives in contracts, contractors not having to maintain the code at their own cost, and immature acquisition practices are potential explanations for the lower Transferability and Changeability scores on government applications. Regardless of the cause, Figure 12 indicates that when COBOL applications are removed from the sample, government applications have the highest proportion of complex components in the Appmarq sample.

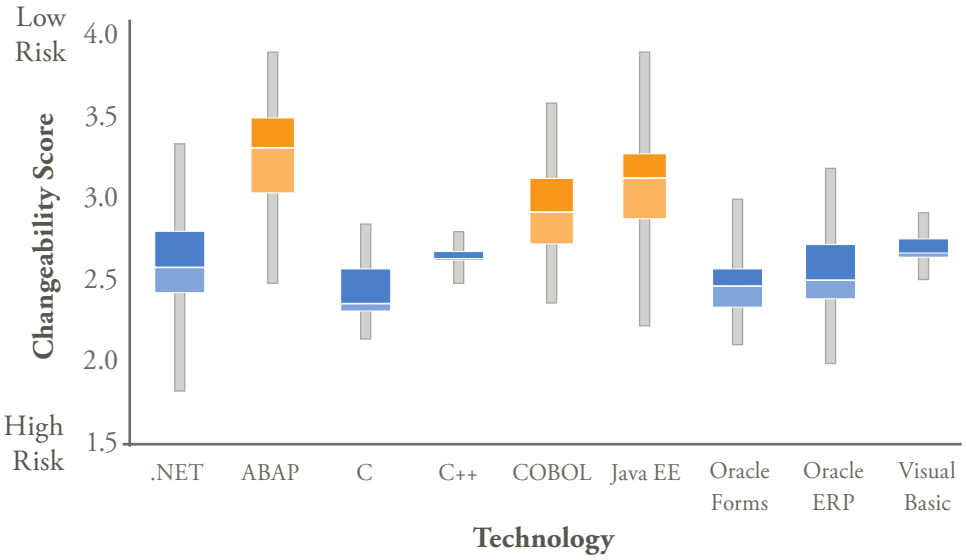
Figure 12. Complexity of Components (Not Including COBOL)



Compared to Transferability scores, the Changeability scores exhibited an even wider distribution indicating that they may be affected by factors other than industry segment. Figure 13 presents Changeability scores by technology type, and shows ABAP, COBOL, and Java-EE had higher Changeability scores than other technologies. It is not surprising that ABAP achieved the highest Changeability scores since most ABAP code customizes commercial off-the-shelf SAP systems.

The lowest Changeability scores were seen in applications written in C, a language that allows great flexibility in development, but apparently sacrifices ease of modification.

Figure 13. Changeability Scores by Technology



PART II: New Insights This Year

Variations in quality are not explained by sourcing model alone

Finding 5—No Structural Quality Difference Due to Sourcing or Shoring

The Appmarq sample was analyzed based on whether applications were managed by inhouse or outsourced resources. A slightly larger proportion of the applications were developed by outsourced resources (*n*=390) compared to inhouse resources (*n*=355). Figure 14 presents data comparing inhouse and outsourced applications, showing no difference between their Total Quality Index scores. This finding of no significant differences was also observed for each of the individual quality characteristic scores. One possible explanation for these findings is that many of the outsourced applications were initially developed inhouse before being outsourced for maintenance. Consequently, it is not unexpected that their structural quality characteristics are similar to those whose maintenance remained inhouse.

Similar findings were observed for applications developed onshore versus offshore. Most of the applications in the Appmarq sample were developed onshore (*n*=585) even if outsourced. As is evident in Figure 15, no significant differences were detected in the Total Quality Index between onshore and offshore applications. There were also no differences observed among each of the individual quality characteristic scores.

Figure 14. Total Quality Index Scores for Inhouse vs. Outsourced

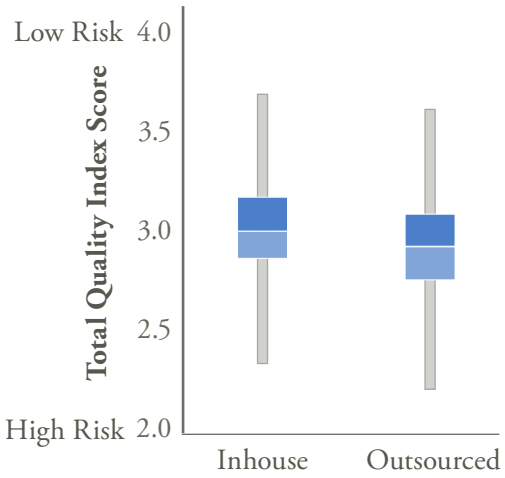
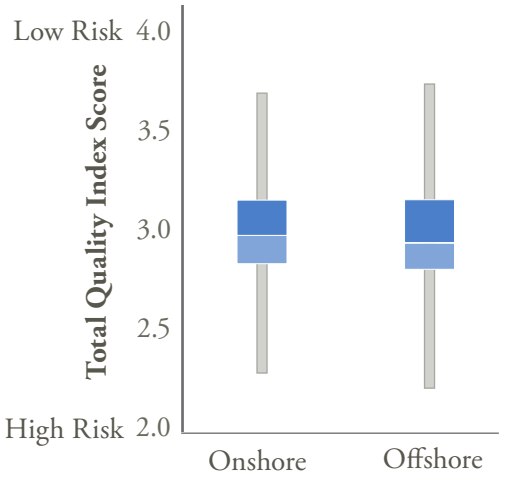


Figure 15. Total Quality Index Scores for Onshore vs. Offshore



Quality lowest with custom development methods, and Waterfall scores highest in Changeability and Transferability

Finding 6—Development Methods Affect Structural Quality

The five quality characteristics were analyzed for differences between the development method used on each of the applications. For the 204 applications that reported their development method, the most frequently reported methods fell into four categories: agile/iterative methods (*n*=63), waterfall (*n*=54), agile/waterfall mix (*n*=40), and custom methods developed for each project (*n*=47). As is evident in Figure 16a, scores for the Total Quality Index were lowest for applications developed using custom methods rather than relying on a more established method. Similar trends were observed for all of the quality characteristics except Transferability.

The Transferability and Changeability scores for applications that reported using waterfall methods are higher than those using agile/iterative methods, as displayed in Figures 16b and 16c. This trend was stronger for Changeability than for Transferability. In both cases, the trend for a mix of agile and waterfall methods was closer to the trend for agile than to the trend for waterfall.

It appears from these data that applications developed with agile methods are nearly as effective as waterfall at managing the structural quality affecting business risk (Robustness, Performance, and Security), but less so at managing the structural quality factors affecting cost (Transferability and Changeability). The agile methods community refers to structural quality as managing Technical Debt, a topic we will discuss in Part III.

Figure 16a. Total Quality Index Scores by Development Methods

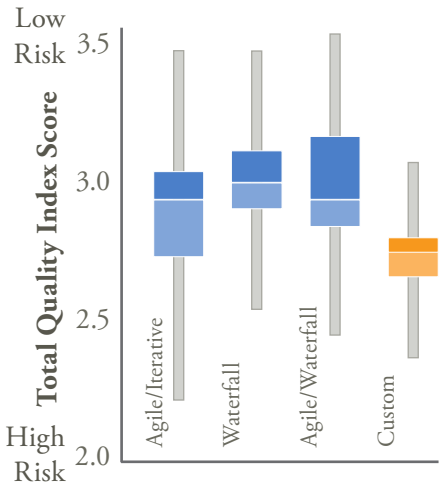


Figure 16b. Transferability Scores by Development Methods

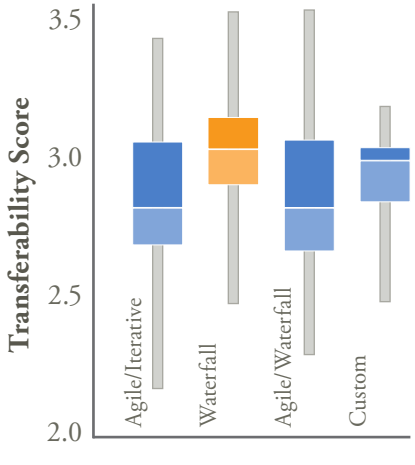
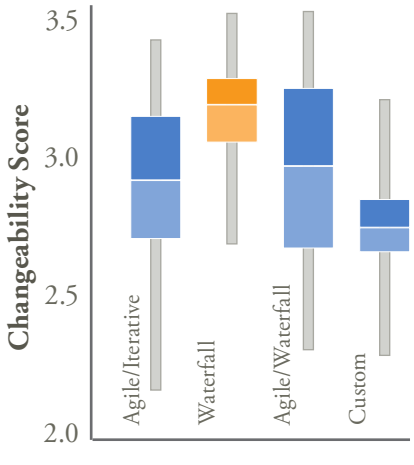


Figure 16c. Changeability Scores by Development Methods



Development Methods

Finding 7— Scores Decline with More Frequent Releases

The five quality characteristics were analyzed based on the number of releases per year for each of the applications. The 319 applications that reported the number of releases per year were grouped into three categories: one to three releases (*n*=140), four to six releases (*n*=114), and more than six releases (*n*=59). As shown in Figure 17a, 17b, and 17c, scores for Robustness, Security, and Changeability declined as the number of releases grew, with the trend most pronounced for Security. Similar trends were not observed for Performance and Transferability. In this sample most of the applications with six or more releases per year were reported to have been developed using custom methods, and the sharp decline for projects with more than six releases per year may be due in part to less effective development methods.

Figure 17a. Robustness Scores by Number of Releases per Year

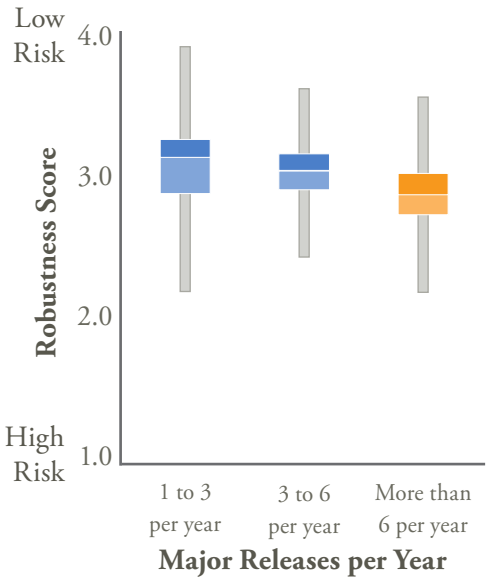


Figure 17b. Security Scores by Number of Releases per Year

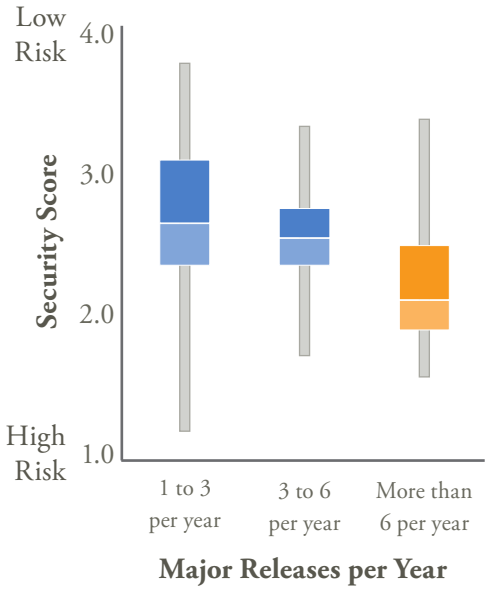
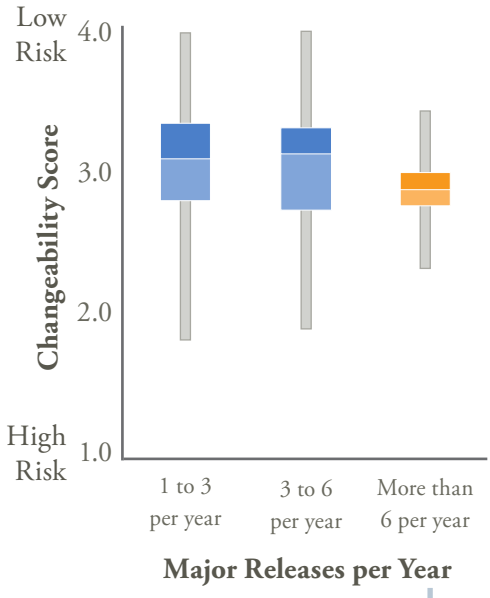


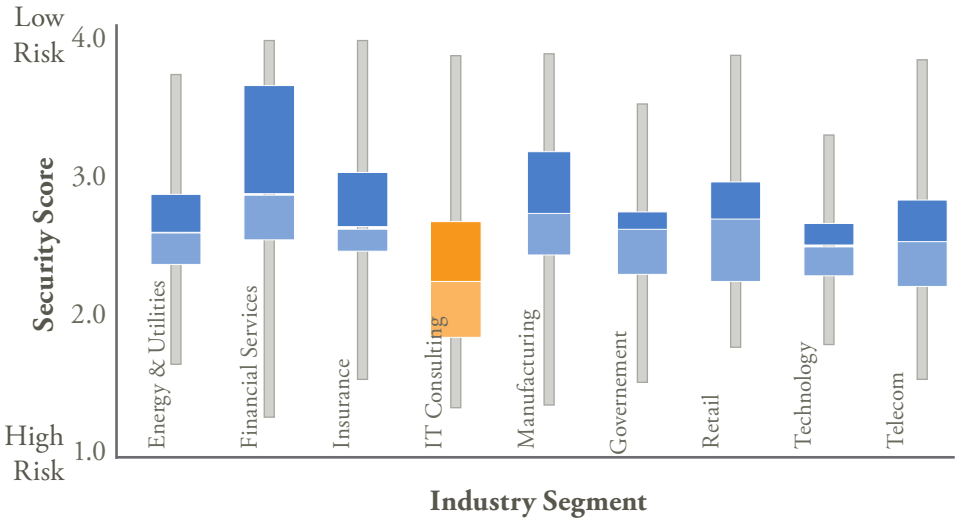
Figure 17c. Changeability Scores by Number of Releases per Year



Finding 8—Security Scores Lowest in IT Consulting

As is evident in Figure 18, Security scores are lower in IT consulting than in other industry segments. These results did not appear to be caused by technology, since IT consulting displayed one of the widest distributions of technologies in the sample. Deeper analysis of the IT consulting data indicated that the lower Security scores were primarily characteristic of applications that had been outsourced to them by customers. In essence, IT consulting companies were receiving applications from their customers for maintenance that already contained significantly more violations of good security practices.

Figure 18. Security Scores by Industry Segment



Finding 9—Maintainability Declines with Number of Users

The five quality characteristics were analyzed to detect differences based on the number of users for each of the 207 applications that reported usage data. Usage levels were grouped into 500 or less (*n*=38), 501 to 1000 (*n*=43), 1001 to 5000 (*n*=26), and greater than 5000 (*n*=100). Figures 19a and 19b show scores for Transferability and Changeability rose as the number of users grew. Similar trends were not observed for Robustness, Performance, or Security. A possible explanation for these trends is that applications with a higher number of users are subject to more frequent modifications, putting a premium on Transferability and Changeability for rapid turnaround of requests for defect fixes or enhancements. Also, the most mission critical applications rely on most rigid (waterfall like) processes.

Figure 19a. Transferability by Number of Application Users

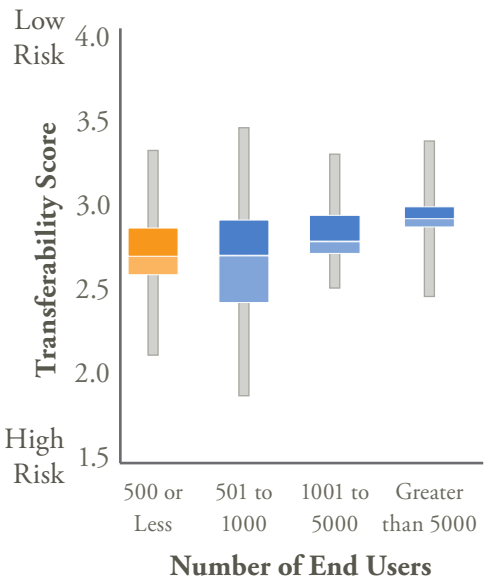
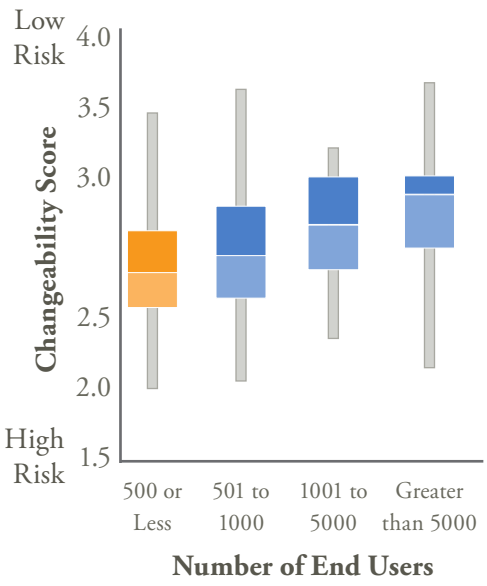


Figure 19b. Changeability by Number of Application Users



PART III: Technical Debt

This report takes a very conservative approach to quantifying Technical Debt

Finding 10—Average $3.61 of Technical Debt per LOC

Technical Debt represents the effort required to fix problems that remain in the code when an application is released. Since it is an emerging concept, there is little reference data regarding the Technical Debt in a typical application. The CAST Appmarq benchmarking repository provides a unique opportunity to calculate Technical Debt across different technologies, based on the number of engineering flaws and violations of good architectural and coding practices in the source code. These results can provide a frame of reference for the application development and maintenance community.

Since IT organizations will not have the time or resources to fix every problem in the source code, we calculate Technical Debt as a declining proportion of violations based on their severity. In our method, at least half of the high severity violations will be prioritized for remediation, while only a small proportion of the low severity violations will be remediated. We developed a parameterized formula for calculating the Technical Debt of an application with very conservative assumptions about parameter values such as the percent of violations to be remediated at each level of severity, the time required to fix a violation, and the burdened hourly rate for a developer. This formula for calculating the Technical Debt of an application is presented on the following page.

To evaluate the average Technical Debt across the Appmarq sample, we first calculated the Technical Debt per line of code for each of the individual applications. These individual application scores were then averaged across the Appmarq sample to produce 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 LOC, and applications of 300,000 or more LOC carry more than $1 million of Technical Debt ($1,083,000). The cost of fixing Technical Debt is a primary contributor to an application's cost of ownership, and a significant driver of the high cost of IT.

This year's Technical Debt figure of $3.61 is larger than the 2010 figure of $2.82. However, this difference cannot be interpreted as growth of Technical Debt by nearly one third over the past year. This difference is at least in part, and quite probably in large part, a result of a change in the mix of applications included in the current sample.

Technical Debt Calculation

Our approach for calculating Technical Debt is defined below:

1. The density of coding violations per thousand lines of code (KLOC) is derived from source code analysis using the CAST Application Intelligence Platform. The coding violations highlight issues around Security, Performance, Robustness, Transferability, and Changeability of the code.

2. Coding violations are categorized into low, medium, and high severity violations. In developing the estimate of Technical Debt, it is 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.

3. To be conservative, we assume that low, moderate, and high severity problems would each take one hour to fix, although industry data suggest these numbers should be higher and in many cases is much higher, especially when the fix is applied during operation. We assumed developer cost at an average burdened rate of $75 per hour.

4. Technical Debt is therefore calculated using the following formula: Technical Debt = (10% of Low Severity Violations + 25% of Medium Severity Violations + 50% of High Severity Violations) * No. of Hours to Fix * Cost/Hr.

Only one third of Technical Debt carries immediate business risks

Finding 11— Majority of Technical Debt Impacts Cost and Adaptability

Figure 20 displays the amount of Technical Debt attributed to violations that affect each of the quality characteristics. Seventy percent of the Technical Debt was attributed to violations that affect IT cost: Transferability and Changeability. The other thirty percent involved violations that affect risks to the business: Robustness, Performance, and Security.

Similar to the findings in the complete sample, in each of the technology platforms the cost factors of Transferability and Changeability accounted for the largest proportion of Technical Debt. This trend is shown in Figure 21, which displays the spread of Technical Debt across quality characteristics for each language. However, it is notable that the proportion of Technical Debt attributed to the three characteristics associated with risk (Robustness, Performance, and Security) is much lower in C, C++, COBOL, and Oracle ERP. Technical Debt related to Robustness is proportionately higher in ABAP, Oracle Forms, and Visual Basic.

Figure 20. Technical Debt by Quality Characteristics for the Complete Appmarq Sample

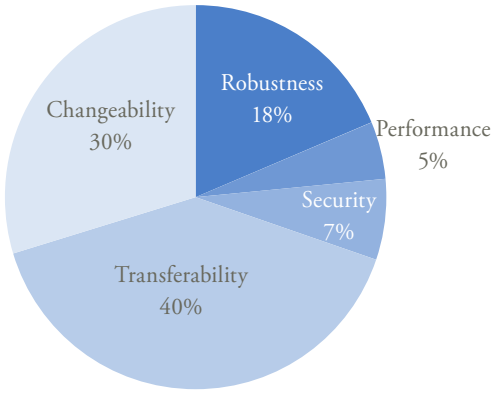
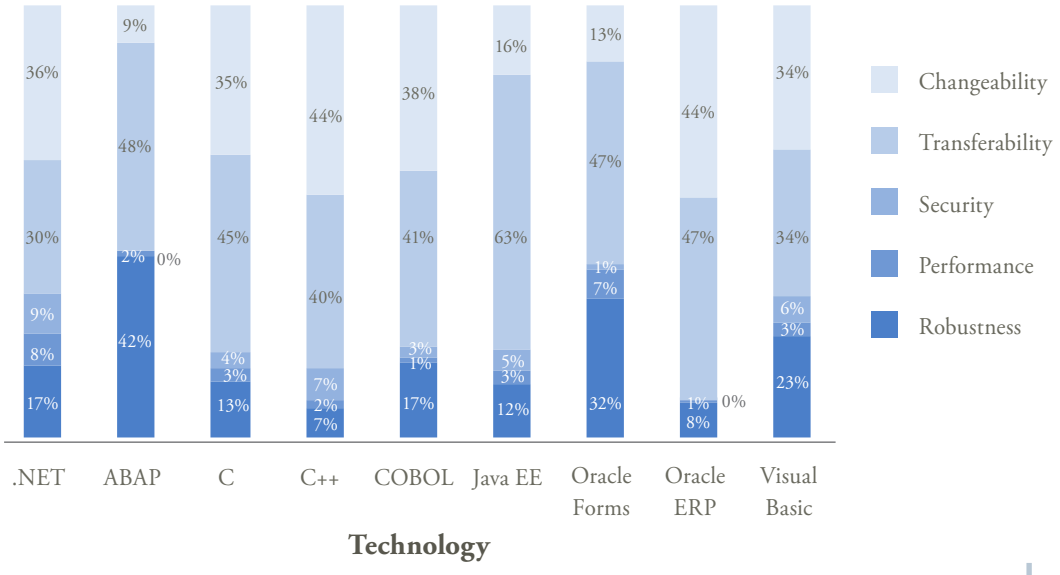


Figure 21. Technical Debt by Quality Characteristics for Each Language



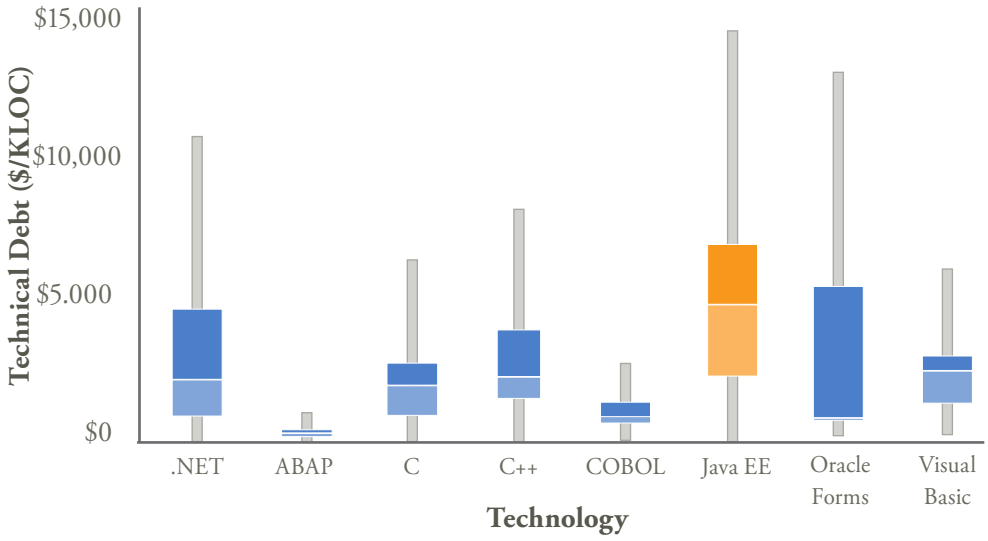
Finding 12—Technical Debt is Highest in Java-EE

Technical Debt was analyzed within each of the development technologies. As shown in Figure 22, Java-EE had the highest Technical Debt scores, averaging $5.42 per LOC. Java-EE also had the widest distribution of Technical Debt scores, although scores for .NET and Oracle Forms were also widely distributed. COBOL and ABAP had some of the lowest Technical Debt scores.

Future Technical Debt Analyses

The parameters used in calculating Technical Debt can vary across applications, companies, and locations based on factors such as labor rates and development environments. During the past two years we have chosen parameter values based on the previously described conservative assumptions. In the future, we anticipate changing these values based on more accurate industry data on average time to fix violations and strategies for determining which violations to fix. The Technical Debt results presented in this report are suggestive of industry trends based on the assumptions in our parameter values and calculations. Although different assumptions about the values to set for parameters in our equations would produce different cost results, the relative comparisons within these data would not change, nor would the fundamental message that Technical Debt is large and must be systematically addressed to reduce application costs, risks, and adaptability.

Figure 22. Technical Debt within Each Technology



Concluding Comments

The findings in this report establish differences in the structural quality of applications based on differences in development technology, industry segment, number of users, development method, and frequency of release. However, contrary to expectations, differences in structural quality were not related to the size of the application, whether its development was onshore or offshore, and whether its team was internal or outsourced. These results help us better understand the factors that affect structural quality and bust myths that lead to incorrect conclusions about the causes of structural problems.

These data also allows us to put actual numbers to the growing discussion of Technical Debt—a discussion that has suffered from a dearth of empirical evidence. While we make no claim that the Technical Debt figures in this report are definitive because of the assumptions underlying our calculations, we are satisfied that these results provide a strong foundation for continuing discussion and the development of more comprehensive quantitative models.

We strongly caution against interpreting year-on-year trends in these data due to changes in the mix of applications making up the sample. As the Appmarq repository grows and the proportional mix of applications stabilizes, with time we will be able to establish annual trends and may ultimately be able to do this within industry segments and technology groups. Appmarq is a benchmark repository with growing capabilities that will allow the depth and quality of our analysis and measurement of structural quality to improve each year.

The observations from these data suggest that development organizations are focused most heavily on Performance and Security in certain critical applications. Less attention appears to be focused on removing the Transferability and Changeability problems that increase the cost of ownership and reduce responsiveness to business needs. These results suggest that application developers are still mostly in reaction mode to the business rather than being proactive in addressing the long term causes of IT costs and geriatric applications.

Finally, the data and findings in this report are representative of the insights that can be gleaned by organizations who establish their own Application Intelligence Centers to collect and analyze structural quality data. Such data provide a natural focus for the application of statistical quality management and lean techniques. The benchmarks and insights gained from such analyses provide excellent input for executive governance over the cost and risk of IT applications.

CAST Research Labs

CAST Research Labs (CRL) was established to further the empirical study of software implementation in business technology. Starting in 2007, CRL has been collecting metrics and structural characteristics from custom applications deployed by large, IT-intensive enterprises across North America, Europe and India. This unique dataset, currently standing at approximately 745 applications, forms a basis to analyze actual software implementation in industry. CRL focuses on the scientific analysis of large software applications to discover insights that can improve their structural quality. CRL provides practical advice and annual benchmarks to the global application development community, as well as interacting with the academic community and contributing to the scientific literature.

As a baseline, each year CRL will be publishing a detailed report of software trends found in our industry repository. The executive summary of the report can be downloaded free of charge by clicking on the link below. The full report can be purchased by contacting the CAST Information Center at +1 (212) 871-8330 or visit:
http://research.castsoftware.com.

Authors

Jay Sappidi, Sr. Director, CAST Research Labs
Dr. Bill Curtis, Senior Vice President and Chief Scientist, CAST Research Labs
Alexandra Szynkarski, Research Associate, CAST Research Labs

CAST
ACHIEVE INSIGHT. DELIVER EXCELLENCE.

For more information, please visit **research.castsoftware.com**