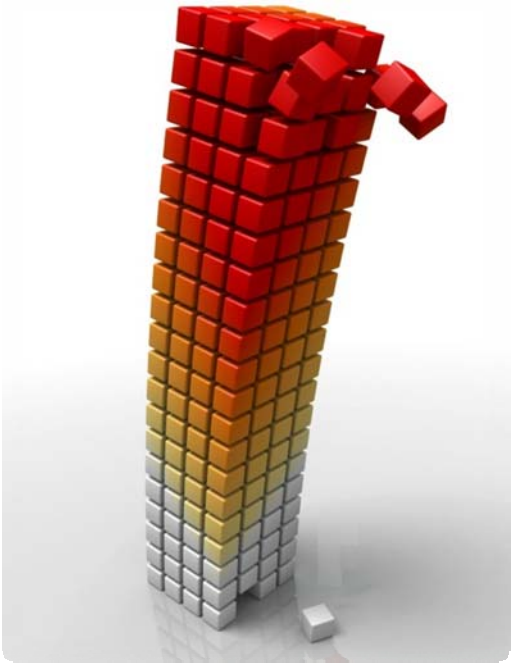
How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations

Pragmatic recipes to ensure the full compliance of custom-built IT systems with rules and measures of good software practice from CISQ, the Consortium for IT Software Quality
WWW.IT-CISQ.ORG



This paper explains what steps need to be taken in order to deliver secure, efficient, reliable, and easy-to-change complex IT systems, with coding and architecture that comply with CISQ recommendations and emerging standards. Based on twenty years of research in software engineering & business IT, CISQ non-functional requirements (Reliability, Performance Efficiency, Security and Maintainability) are at the core of the CISQ standards & recommendations. The paper highlights the lack of correlation between good coding practices at the code unit level and value for the business.[1,2,3] It explains and illustrates the technical reasons for how a software application made from myriad high-quality constituents can turn into a fragile, unpredictable, and dangerous system, occasionally disrupting a vital business process. It includes strong technical proof points supporting the need for a system-level, architectural analysis of source code and applications' inner structure to deliver high-quality business applications. Finally, it gives a quick introduction to the software measurement and analysis solutions available on the market.

CIOs, CTOs, Enterprise Architects, Application Owners, and all IT executives who want to rapidly lower production defects and improve end-user satisfaction and business productivity will understand what should be the next step toward an effective software quality policy.

The ideas in this paper and the CISQ Quality Characteristic measures it describes result from contributions from many sources—from practicing software engineers; PhD scientists at R&D laboratories; data from large IT organizations; software assurance work at MITRE Corp.; standards from OMG and its special interest group the Consortium for IT Software Quality (CISQ); the US National Institute of Standards and Technology (NIST); and decades of research published by IEEE, ACM, the journals Empirical Software Engineering and Software: Evolution and Process, and the Software Engineering Institute at Carnegie Mellon University.

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

Dr. Richard Mark Soley is the Chairman and CEO of the Object Management Group and an industry luminary who is keen on educating the IT market on the importance of measuring and improving the quality of its software. As CEO, Dr. Soley is ultimately responsible for all of the business of the OMG, including Board activities and oversight of the OMG's neutral, international and open Technical Process for producing industry standards. Dr. Soley serves as a valuable resource for a broad range of topics ranging from predictions and trends in the industry to the details of CORBA and UML (based on his leadership as the OMG's original Technical Director for the consortium's first eight years). Dr. Soley is a co-founder of the Consortium for IT Software Quality (CISQ) and gives frequent keynotes on the importance of its measurement standards for industry and government software.

Today's Enterprise IT App and Its Software Quality Challenges

Today's enterprise applications are made of multiple layers incorporating different components, software frameworks, heterogeneous technologies, and different languages; whereas 30 years ago a single platform and one language often covered all of these aspects. A massive wave in mobile, distributed & cloud computing is making applications even more complicated, and software development & integration is likely to get much more complex. Add to this the fact that most of today's enterprise systems are an assembly of old legacy software with newly developed application code interacting with software packages from different vendors using different standards. Ultimately, what we find supporting mission critical business processes is a sophisticated, but extremely complex stack of technologies integrated into what we euphemistically call a software 'product' for which there is no overall design nor architecture.

A typical customer support management or billing system can consist of more than four million lines of code written in four or five different programming languages, forming 40,000+ program units, all interconnected and interacting with many giant data structures. If printed out such a system would be 30 feet thick, written in Chinese, English, German, French by the different development sites, telling a very complex story, with tens of thousands of inter-dependent components occasionally colliding with each other.

To ensure that the entire system is safe, robust, efficient, and easy to maintain, engineering disciplines make clear distinctions between the different analysis levels required. These distinctions are necessary to ensure that these quality attributes are analyzed from every perspective that could affect the operational performance of an application. In software, from the very basic to the most sophisticated, these levels would be the following:

- First, ***UNIT LEVEL ANALYSIS*** represents the ability to analyze ***a single unit of code***. Depending on the technology, a unit of code can be a method, a function, or an entire program for unstructured languages.

- Second, ***TECHNOLOGY LEVEL ANALYSIS*** is the ability to analyze ***an integrated collection of code units written in the same language***, by taking into account the dependencies across programs, components, files, or classes. Technology Level analysis is typically performed at the level of a layer in the application and can be described as an 'intra-technology' analysis, meaning within a single technology set. Thus, the analysis of a modern business application would include a collection of technology level analyses for each of the different technologies integrated into the full application.

- Third, ***SYSTEM LEVEL ANALYSIS*** would refer to the ability to analyze ***all the different code units and different layers of technology to get a holistic view of the entire integrated business application***. System level analysis allows us to visualize complete transaction paths from user entries, through user authentication and business logic, down to sensitive data access. System level analysis allows us to detect

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

violations of good architectural practice as well as dysfunctional interactions among different technologies that could cause operational outages, security breaches, data corruption, and other problems.

A long awaited software quality standard for IT business applications has been recently published by the Consortium for IT Software Quality, a consortium co-sponsored by the Object Management Group and the Software Engineering Institute at Carnegie Mellon University.[4] This consortium's 24 member companies, including many Fortune Global 200 Companies, have defined automated quality measures four technical characteristics as well as the underlying rules of good architectural and coding practice that must be checked to provide the inputs for these measures. They have classified the software engineering best practices into two main categories: rules of good coding practice within a program at the Unit Level without the full Technology or System Level context in which the program operates, and rules of good architectural and design practice at the Technology or System level that take into consideration the broader architectural context within which a unit of code is integrated. Figure 1 displays examples of these rules at the Unit and Technology/System Levels.

Characteristic	Good Coding Practices @ Unit-Level	Good Architectural Practices at Technology/System Levels
RELIABILITY	Protecting state in multi-threaded environments Safe use of inheritance and polymorphism Resource bounds management, Complex code Managing allocated resources, Timeouts, Built-in remote addresses	Multi-layer design compliance Software manages data integrity and consistency Exception handling through transactions Class architecture compliance
PERFORMANCE EFFICIENCY	Compliance with Object-Oriented best practices Compliance with SQL best practices Expensive computations in loops Static connections versus connection pools Compliance with garbage collection best practices	Appropriate interactions with expensive or remote resources Data access performance and data management Memory, network and disk space management Centralized handling of client requests Use of middle tier components versus procedures and database functions
SECURITY	Use of hard-coded credentials Buffer overflows Broken or risky cryptographic algorithms Missing initialization Improper validation of array index Improper locking References to released resources Uncontrolled format string	Input validation SQL injection Cross-site scripting Failure to use vetted libraries or frameworks Secure architecture design compliance
MAINTAINABILITY	Unstructured and Duplicated code High cyclomatic complexity Controlled level of dynamic coding Over-parameterization of methods Hard coding of literals Excessive component size Compliance with OO best practices	Compliance with initial architecture design Strict hierarchy of calling between architectural layers Excessive horizontal layers

Fig 1. Elements of the CISQ Quality Characteristic Measures[5].

Correlations between programming mistakes and production defects unveil something really intriguing and, to some extent, counter-intuitive.[6,7] ***It appears that basic Unit Level errors account for 92% of the total errors in the source code.[8,9,10] These numerous code level issues eventually count for only 10% of the defects in production. On the other hand, bad software engineering practices at the Technology and System Levels account for only 8% of total defects, but consume over half the effort spent on fixing problems, and eventually lead to 90% of the serious reliability, security, and efficiency issues in production.[11,12]*** This means that tracking and fixing bad programming practices at the Unit Level alone may not translate into the anticipated business impact, since many of the most devastating defects can only be detected at the Technology and System Levels. Tracking these Technology and System Level programming

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

errors could save more than half of the rework during the building phases, while drastically decreasing the production incident rate.

The logical and obvious conclusion is to dramatically increase the effort focused on detecting the few really dangerous architectural software defects. Unfortunately, identifying such 'architecturally complex violations' is anything but easy. It requires holistic analysis at both the Technology and System Levels, as well as a comprehensive, detailed understanding of the overall structure and layering of an application. For those needing further confirmation and explanation of such problems, the most common examples for each of the four CISQ characteristics, are described below, with greater detail provided in the technical addendum at the end of this report. The four CISQ quality characteristics were defined to be consistent with ISO/IEC 25010, which replaces ISO/IEC 9126.

#1 Reliability & Resiliency: Lack of reliability and resilience is often rooted in the "*error handling*." Local, Unit Level analysis can help find missing error handling when it's related to local issues, but when it comes to checking the consistency of the error management across multiple technology stacks, which is tactically always the case in sophisticated business applications, a contextual understanding at the Technology and System Levels is needed. Failure to properly manage error-handling leads to outages and similar problems.

Data corruption also requires the ability at the System level to check the types and structures of data flowing from one language to another or from one software layer to another. A full analysis of the application is mandatory because developers may simply bypass data manipulation frameworks, approved access methods, or layers. As a result, multiple programs may touch the data in an uncontrolled, chaotic way. Other dangerous cases are related to implicit conversions or when a string coming from a JEE frontend is truncated by a COBOL program running on a Mainframe in the backend. Specifically for Reliability, bad coding practices at the Technology Level lead to two-thirds of the serious problems in production.

#2 Performance Efficiency: Performance or efficiency problems are well known to damage end-user productivity, customer loyalty, and to consume more IT resources than they should. The **'**remote calls inside loops' (i.e. remote programs executed on a remote device from another program itself located in a loop) are a well-known example that creates performance problems. A top down, System Level analysis is required to search down the entire system calling graph to identify the source of the problem. When not detected, this type of issue could result in slow response times for all end-users impacted by the over-consumption of CPU, notwithstanding the useless consumption of MIPS. Another common System Level performance issue relates to applications using SQL statements built dynamically which do not leverage the indexation strategy correctly.

Performance issues in the vast majority of cases reside in System Level architectural problems. The Technology or Unit Levels have a much smaller impact statistically, albeit poor OO programming practices or bad memory management at the code stack level could have disastrous consequences.

#3 Security & Vulnerability: Detecting backdoor or unsecure dynamic SQL queries through multiple layers requires a deep understanding of all the data manipulation layers as well as the data structure itself. Overall, security experts Greg Hoglund and Gary McGraw believe cross-layer security issues account for 50% of all the security issues.[13] Yet, whatever the percent, security is not something that tolerates much approximation.

The software assurance community has identified many sources of security problems that are described in detail in the Common Weakness Enumeration repository (cwe.mitre.org) maintained by Mitre Corp. This repository was used in defining the violations incorporated into CISQ's automated measure for Security.

#4 Maintainability, Adaptability & Changeability: Many believe Maintainability is primarily about readability of the code and thorough documentation. While it is correct to say code tidiness is important, approximately half of the root causes of poor Maintainability are caused by failures to observe design and architectural rules. Architects unanimously agree that it tends to create a 'spaghetti monster', where the smallest change may have unpredictable and devastating consequences.

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

The Right Move

"The whole is more than the sum of its parts" – Aristotle.

Aristotle's concept is that the quality of the whole has little to do with the sum of the qualities of each individual component. In the software engineering of large applications, quite obviously, structural quality is far more than an intrinsic property of coded components.[12,13] The exact same piece of code can be excellent in quality or highly dangerous, depending on the context in which it operates. As in all other disciplines, it doesn't make sense to bother developers with programming efficiency in the exact same way when they're developing a program dealing with millions of customer names, versus a piece of code dealing with a handful of parameters. The rather simple analogy between brick-and-mortar building architectures and software architecture works well. The structural quality of a building is a combination of the quality of the bricks, the quality of the laying and mortaring of the bricks in the wall, the quality of the framing of the building, and the quality of the assembly of all these components together. The best gold bricks on earth, if poorly assembled, won't make a structurally stable building.

Having said that, what is the right move to deliver robust, resilient, secure and, efficient business critical systems in line with CISQ recommendations?

Hiring the best, most talented developers available on the market is always a good step, and improving process maturity will help, for sure. But none of this will 'guarantee' the quality of the product itself. As in any other industry, a complete structural quality check at the System Level, preferably all along the manufacturing chain, must be performed. The following are solutions offered by the market for automated verification of programming best practices, at the Unit, Technology and System Levels.

At the **Unit Level**, most of the coding practices can be checked with code analyzers already embedded into current development environments or with native features in the IDE, such as those offered in Microsoft Visual Studio or Eclipse. There are also some useful tools available on the market which target individual developers and typically run on developers' workstations. The most popular are developed by the Open Source communities (PMD, Checkstyle, etc.). These tools capture dead unused variables, overcomplicated expressions, and similar items useful for the hygiene of the code. Such tools tend to drown developers in myriad coding errors, sometime irrelevant due to the lack of context, but at least they ensure a decent level of readability and tidiness of the code, which is important for its maintainability, and to some extent for code reliability and performance efficiency.

The more serious problems start at the **Technology Level**. There are advanced open source products such as FindBugs for Java that can transcend the Unit Level scope. There are also a few commercial products, such as analyzers for C++ from Grammatech, Klockwork, CAST, and Coverity; analyzers for COBOL from Raincode or Microfocus; and the Sonar portal which aggregates open-sources and commercial products. There are also a few more specialized products such as HP/Fortify or from IBM, that focus on Security and can understand context as long as all code is programmed using the same language. Such products provide interesting value in a single technology context, typically for a monolithic C or COBOL batch program, the interface of a web-based customer-facing app, or for a monolithic Java app with limited GUI and data access.

Finally, to address analysis at the **System Level,** a tempting homemade solution may consist of putting the above mentioned code checkers into one place to try and 'take a picture of the app'. Unfortunately, none of these code analyzers have been designed to talk to each other in a way that would provide understanding of the system's full context and inner-workings. Code analyzers must be able to exchange information and dialogue between each other to eventually create a comprehensive logical image of the app's inner structure, traversing the entire app from end-user entries to data access. Only then can adherence to good coding practices and architectural standards can be measured. Only a handful of companies provide System Level

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

solutions, most notably ASG which focuses mostly on legacy modernization and Cobol/Mainframe environments with its Becubic product and CAST Software which covers a broader set of technologies and languages, supporting the analysis of multi-language, cross-technology portfolios. These products can analyze an independent piece of code or component in stand-alone Unit Level mode, but will require the entire set of code source, script, and database structure to analyze the entire app the System Level. Consequently, they are typically used at the build phase, and can also be integrated into the build chain to provide ongoing, continuous training and visibility across entire app to those developing coded units in its various technology layers.

With all this in mind, **where do we start?** It is actually heavily dependent on the types of systems involved. For embedded systems or monolithic apps, Technology Level analyzers will be enough in most cases since the majority of the system is developed in one language. For complex business, transactional apps, with a predominant data component, there is no other option than to start with System Level analysis. And as in any other industry, a structural quality 'checkpoint' should take place as soon as the product is ready for assessment—at the end of every run, at every build, and at least at the end of the development phase just prior to integration testing. The smart approach might also involve positioning such checkpoints as an IV&V to measure non–functional software quality characteristics such as those defined by CISQ. Because of all the benefits the business can gain from reliable, secure, easy to maintain, and high performing apps, the 'CISQ-checked' label applied to the most critical business systems will resonate quite well.

References

1. Jackson, D., Thomas, M., and Millett, L.I. (2007, Eds.) *Software for Dependable Systems: Sufficient Evidence?* Washington, DC: National Academies Press.

2. Jackson, D. A direct path to dependable software. *Communications of the ACM, 52* (4), 77-88.

3. Jackson, D. (2006). Dependable software by design. *Scientific American*, May 2006.

4. CISQ (2012). CISQ Specifications for Automated Software Quality Measures. Needham, MA Object Management Group, Consortium for IT Software Quality. www.it-cisq.org

5. CISQ (2012) Using Software Measurement in SLAs: Integrating CISQ Structural Quality Measures into Contractual Relationships. Needham, MA Object Management Group, Consortium for IT Software Quality. www.it-cisq.org

6. Curtis, B., Sapiddi, J., Syznkarski, A. (2012). *CAST Report on Application Software Health 2011/2012 (CRASH)*. New York: CAST.

7. Jones, C. & Bonsignour, O. (2012). *Economics of Software Quality.* Boston: Addison-Wesley.

8. Li, et al. (2011). Characteristics of multiple component defects and architectural hotspots: A large system case study. *Empirical Software Engineering, 16* (5), 667-702.

9. Leszak, M., et al. (2000). A case study of root cause defect analysis. *Proceedings of the 22nd International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society, 428-437.

10. Kristiansen, *Software Defect Analysis: An Empirical Study of Causes and Costs in the IT Industry*. NTNU.

11. Spinellis, D. (2006). *Code Quality*. Boston: Addison-Wesley.

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

12. Nygard, M.T. (2007). Release It! Design and Deploy Production Ready Software. Raleigh, NC: Pragmatic Bookshelf.

13. Hoglund, G. & McGraw, G. (2004). *Exploiting Software: How to Break Code.* Boston: Addison-Wesley.

14. Mitre Corp. (2012). *Common Weakness Enumeration*. mitre.cwe.org

15. Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. (1979). Modern coding practices and programmer performance. *IEEE Computer, 12* (12), 41-49.

Technical Addendum

Going one step further, this technical addendum presents meaningful software engineering examples that correlate the programming practices, as per the CISQ categorization, with the impact on business. The four CISQ software characteristics (Reliability, Performance Efficiency, Security, and Maintainability) can be directly related to the business objectives of reducing business risk and IT cost.

1. Reliability — Resiliency & Dependability

As defined by the CISQ, Reliability measures the risk of potential application failures and the stability of an application when confronted with unexpected conditions. The reason for checking and monitoring Reliability is to reduce and prevent application downtime, application outages and errors that directly affect the user's and the company's business performance. From an end-user and business standpoint, reliability is likely to cause the most costly disruption. Here are a few of examples of those issues:

#1. Handling unforeseen situations: Unforeseen IT infrastructure situations occur every day. For mission critical business systems, the resilience – the ability of withstanding shock without breaking – is one of the most demanded non-functional requirements. It is common to rely on exception or error handling to manage unplanned situations. Yet exception handling is not always the development team's top priority. It does not really deliver an immediate, tangible value to the end-user because the conditions trapped by the exception or the error do not occur under regular conditions of use. Typically, it's the type of thing that gets pushed to tomorrow.

Local, unit-level analysis can help to find a missing error handling when it's related to local issues, such as the well-known 'empty catch block', 'missing to test a returned value', or 'avoid catching Exception or Throwable'. Detecting this type of case does not require the understanding of anything other than the code of the function or method. It can be detected as soon as the relevant code is completed and is even caught directly by a professional development environment.

Coding Errors Impacting the RELIABILITY	Context Required	Business Impact (est.)
Error & Exception handling (Unit Level)	Unit Level	10%
Complexity of algorithms		
Error-prone programming		

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

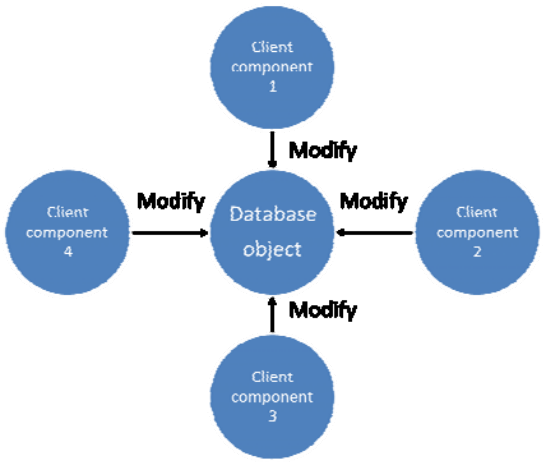
Object-Oriented and Structured Programming best practices (when applicable)	Technology Level	25%
Resource bounds management		
Multi-layer design compliance	System Level	65%
Data integrity and consistency		
Error & Exception handling (across layers)		
Transaction complexity		
Time and state / multi-threading programming		
Null pointers dereference detection		
Resource bounds management		

When it comes to checking the consistency of the error management in an application, the challenge is making sure that every exception thrown receives an appropriate treatment down the chain, across software layer, and so on. The software component raising the exception or error might not be able to react appropriately to the error. For example, if a database insertion fails because the record already exists in the table, the data layer has to report the technical issue (duplicated record), but most of the time it doesn't have the knowledge to decide what should be done next – attempting an update instead of an insert, reporting back to the end-user, or any other handling of the case. It's usually the responsibility of the business logic layer. To enforce such policy, the entire application has to be analyzed at the System Level to determine who catches what and also to check that the right exceptions are thrown and/or caught by the correct layer/library. If not done appropriately, cryptic errors might be reported to users who might then think that the application is broken. Even worse, errors might not be reported at all, letting the end-user believe that the action was successful instead of prompting him/her for an appropriate follow-up. Ultimately, this might create data corruptions as well.

#2. Preventing data corruption: A large part of the source code developed for enterprise business applications is devoted to data handling, and one of the big risks in this instance is related to data corruption. Bad development practices can rapidly lead to erratic behavior in these applications and, worst case scenario, corrupt data. In most (if not all) cases, detecting such potential issues requires the ability to check the structure of the data flow from one language to another, or from one software layer to another. Thus, a full analysis of the application is required.

The following examples related to database access illustrate the situation. Data modifications are usually ruled by the use of specific routines to update/insert/delete a specific API or a data layer that is fully tested to maintain data integrity. The consequence of allowing multiple components to modify data and not make use of the existing tested code is at the origin of many data corruption cases. A System Level analysis will take into consideration all accesses to database carried out by the application components and will check the validity of the different operations. Thus, redundancies that do not respect the application architecture will be detected.

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.



More insidious cases are related to the implicit conversions that occur between two compatible data sources such as the ones found in the different SQL dialects. For example, on MS SQL Server, a string of type CHAR will be implicitly converted to NUMERIC if used in an arithmetic operation. As long as the string contains only numerical characters, no error will be reported. But if for whatever reason an alphanumeric character is part of the string then the further arithmetic operation will fail resulting in a runtime error.

In the context of a C++ business logic layer calling a SQL Server data layer (with stored procedures written in T-SQL), the inconsistency needs to be detected by analyzing both the C++ code and the T-SQL code to establish who calls whom, then inspect the parameters to make sure they are all of identical data type.
There are similar but slightly different issues in COBO that occur when moving a larger variable into a smaller one. COBOL is not a strongly typed language and no error will be reported if a large string coming from a JEE front-end is moved into a smaller area. However, the data will be definitively truncated leading to an invisible but unfortunate data corruption.

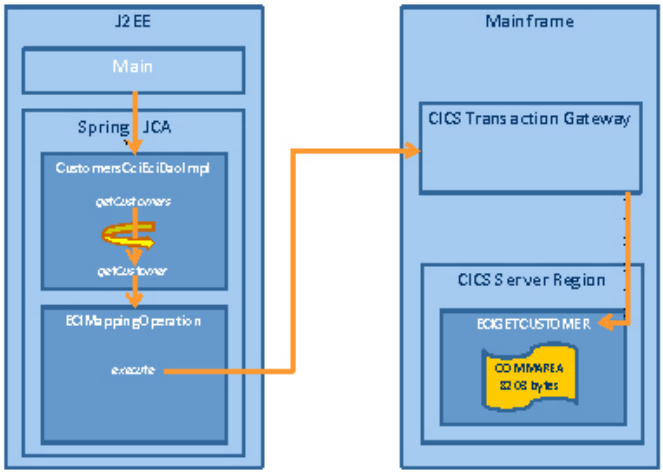
2. Performance Efficiency — Productivity & Customer Loyalty

Performance Efficiency is defined as 'effective operation as measured by a comparison of production with cost (as in energy, time, and money)'. Applied to IT by the CISQ, Performance Efficiency assesses characteristics that affect an application's response behavior and use of resources under stated conditions (ISO/IEC 25010). The analysis of an application's Performance Efficiency attributes indicates risks regarding customer satisfaction, workforce productivity, or application scalability due to response-time degradation. It can also indicate current or projected future inefficient use of processing or storage resources based on wasteful violations of good architectural or coding practice. Here are the most common examples and the good code analysis strategy to prevent them.

#1. Efficient interaction with expensive resources: Studies done after major performance degradations have highlighted an anti-pattern that can be best described as the "remote calls inside loops", where remote means that the calls will be executed on a remote server - web service, database, file system.[12] More precisely the post-mortem analyses of performance related crashes have shown that the root cause of such failures is 'buried' calls to external resources (such as a CICS transaction or a costly SQL access) done inside loops. These calls are difficult to accurately identify at the Unit or Technology Levels. When looking at a loop in a Java or C# code, one can only view a simple method call. Moreover, most of the time, the costly resource is not directly called in the loop. The actual call might be performed several levels down the call graph of the call made in the loop. So if the code analysis stops at the loop stage, the problem won't be trapped and this piece of code will be declared of good quality. Only further research down the call graph would allow for the identification that the method call is, in fact, an access to an expensive resource such as a CICS transaction or a costly SQL access. A

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

concrete example of this anti-pattern between a Java layer and a mainframe back end through a CICS transaction is illustrated below:



the COBOL program return 8208 bytes to the J2EE application for each

The detection of such an anti-pattern requires a System Level analysis of the COBOL layer and the calling layer in Java, communicating with the COBOL through CICS. The COBOL code is analyzed in order to eventually compute the size of the COMMAREA (i.e. the buffer that will be used in CICS transaction) and the Java code is analyzed in order to detect CICS transaction calls that use a large set of data and also take place in a loop. If not detected, this type of issue could result in slow response time not only for the end-user who has initiated the transaction but also the rest of the end-users impacted by the over consumption of CPU due to the slow transaction. Another side effect could be a too important consumption of MIPS leading to an additional cost of running the mainframe.

#2. Efficient accesses to large volumes of data: Big potential performance issues are quite common in applications using relational databases and SQL built dynamically in runtime mode. Such problems are sometimes detected during the costly load-testing phase when it is difficult to fully simulate the operational environment, or more often in production months after the development when the volume of the data increases. Most of the time the problem is due to a SQL query not leveraging the indexation strategy (for example a query not using any indexed column in its WHERE clause). Such an issue is quite complex to detect due to the nature of Dynamic SQL, the fact that the hints of the performance failure are spread across components and layers including the database itself and the size of the data it contains. Indeed, a SQL query that no index can support is not an issue on a small set of data. It becomes an app killer if the data size is large. And again such a performance anti-pattern requires a comprehensive System Level analysis of the client language (Java, .NET, ABAP, etc.), the analysis of the structure of the database including table indexes and table size, and lastly the ability to build the potential dynamic SQL statement that no index could support.

Coding Errors Impacting the EFFICIENCY	Context Required	Business Impact
Compliance with garbage collection best practices	Unit Level	10%
Expensive computations in loops		
Memory, network and disk space management	Technology Level	10%
Compliance with Object-Oriented best practices		
Compliance with SQL best practices		
Appropriate interactions with expensive and/or remote resources	System Level	80%

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

Data access performance and data management		
Centralized handling of client requests		
Use of middle tier components versus stored procedures and database functions		
Algorithm complexity		

3. Security — Identifying Vulnerabilities

As per the CISQ definition, Security assesses the degree to which an application protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization (ISO 25010). As exposed by security experts Greg Hoglund and Gary McGraw in *Exploiting Software – How to Break Code*, "Microsoft reports that around 50% of the problems uncovered after the 2002 security push were design-levels problems."[13] Security issues that require a System Level analysis and investigation of the many different layers and technologies of an application account for 50% of all the security issues, the rest being mostly at the Technology Level. These issues have been captured and described in the Common Weakness Enumeration repository.[14] A couple of examples are presented below.

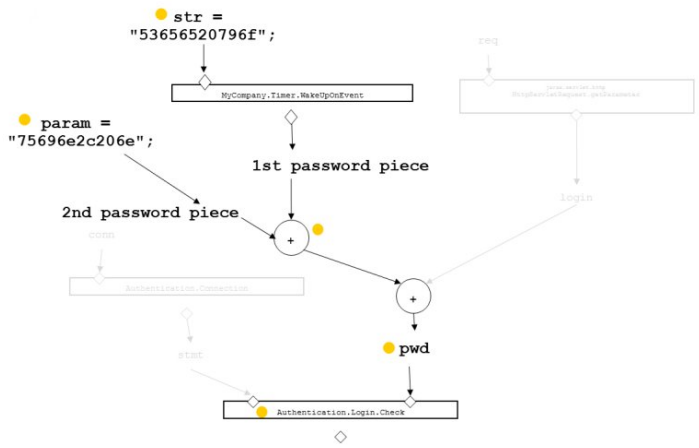
#1. Detecting unsecure dynamic SQL queries thru multiple layers: Data intensive enterprise applications typically manage data update rights through a single mechanism (layer of classes, SQL stored procedures...). When looking at the code that executes an SQL table update in such context, the difference between good and bad coding isn't obvious. Both calls may look exactly the same:

```
ResultSet rs = stmt.executeQuery( myquery )
```

The secure update will have the string *myquery* built with the call of the predefined stored procedures, while the insecure code will just contain a raw SQL update statement. Also the name of the SQL table to be updated can be stored in another string variable in a location far from the method that will run the *executeQuery*. The detection of such security vulnerabilities involves the analysis of all the transactions through the layers of the application, detecting the access to the database and making sure that the actual text sent to the database is the appropriate one. It requires the understanding of how the dynamic SQL is built during the execution of the application, and the understanding of the dataflow to cope with the dynamic aspect of SQL code that is built at execution time. Such quality control can also be applied to the detection of all illegal or unsecure dynamic SQL updates embedded into the client code. Such findings also require an advanced dataflow engine that can track SQL string construction and analysis capabilities across large applications.

#2. Detecting backdoors: Backdoors are another example of security issues that span over multiple locations in application code stacks and require the System Level analysis of the entire code base to be detected. Backdoors require credentials to access the authentication server and the resources of the application. Credentials are usually stored in another part of the application. And in order to keep the backdoor as secret as possible the path between the credentials and the authentication server is made complex. The password, for example, would be stored in multiple string variables located in different classes and files to be concatenated before being sent to the authentication server as shown in the schema below.

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.



To detect a backdoor, looking inside a single code unit or technology layer won't be of any help. Once again, a System Level analysis with additional dataflow capabilities to identify strings variables used for authentication is needed.

Coding Errors Impacting the SECURITY	Context Required	Business Impact
Improper locking	Unit Level	10%
Failure to use vetted libraries or frameworks		
Uncontrolled format string		
Improper validation of array index		
Use of hard-coded credentials	Technology Level	40%
References to released resources		
Cross-site scripting	System Level	50%
Buffer overflows		
SQL injection		
Secure architecture design compliance		

4. Maintainability — Changeability & Adaptability

As defined by CISQ, Maintainability represents the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers (ISO 25010). Maintainability incorporates such concepts as changeability, modularity, understandability, testability, and reusability. Measuring maintainability is important for business or mission-critical applications where an organization must respond rapidly to regulation, customer, or competitor-driven changes. It is also a well-known key to keeping IT costs under control.[15]

Maintainability is usually related to the readability of the code. This can be achieved through adherence to programming practices, including proper documentation leveraging the possibilities offered by languages such as Java or C# to embed structured, documentation-consistent, and meaningful naming conventions, and a clear programming 'style' that is well-structured. Most of these good programming practices are usually

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.

verifiable on a file per file basis, one program at a time, and as such Unit Level analysis is often enough to check these practices, which are done most often by development tools embedded in the IDE.

More crucial for the life expectation of complex, business critical IT systems are the architectural design and the certainty that design decisions have been correctly and consistently applied by the development team all along the app life cycle. Application managers must be vigilant to ensure the structure of the application is and remains sound and healthy, and that the initial design does not morph into a giant spaghetti monster preventing anyone from making the smallest change without undergoing a costly test cycle or generating tons of undetectable regression bugs.

Unit-Level analysis cannot detect these types of problems, which unfortunately are the ones which can block and eventually kill an app. The cost of the cleaning up such problems can be greater than the cost of rewriting everything. Technology Level analysis can help, but only on monolithic applications, and the analysis will not be able to span the different layers of the application. When it comes to ensuring that a modern multi-layer application built with a mix of technologies such as C# for the front-end, JEE for the middle layer, and a SQL RDBMS sustains adherence to its non-functional, structural requirements, only a System Level analysis can help prevent the level of architectural degradation that usually occurs with continued maintenance.

Coding Errors	**Context Required**	**Business Impact**
Cyclomatic complexity	Unit Level	25%
Hard coding of literals		
Excessive prgs size		
Unstructured and Duplicated code	Technology Level	25%
Controlled level of dynamic coding		
Compliance with OO best practices		
Tightly coupled modules	System Level	50%
Strict hierarchy of calling between architectural layers		
Data access performance and data management		
Excessive horizontal layers		
Encapsulated data access		

109 Highland Ave, Needham, MA 02494, U.S.A., Tel: +1-781-444 0404, Fax: +1-781-444 0320, Web: www.omg.org.