Beta General Code Agnostic Quality Standards

The Consular Systems and Technology Division of the Bureau of Consular Affairs recognizes the code quality standards between Bureau systems are inconsistently applied and will ultimately harm its ability to unify systems toward a modern architecture. By addressing this issue with widespread adoption, training, adaptation and monitoring of Division guidelines throughout its family of business systems, tangible operational stability benefits including improved reliability, faster time to repair and resistance to security threats will principally follow suit. Improved system maintainability, performance and management transparency is anticipated when accompanied by corresponding software development life cycle processes and tooling.

This paper is an adaptation of “How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations” produced by Consortium for IT Software Quality (CISQ) which proposes a minimal set of language agnostic code development guidelines. With a framework of language agnostic guidelines, the Division shall endorse a set of detailed and actionable set of coding standards for languages approved for use within the enterprise. These standards are understood to be dynamic documents subject to continuous improvement to capture changes to architecture and Bureau constraints.. Input from development staff and software engineering staff, leveraging both private and public sector personnel were included.

# Introduction

In pursuit of division code quality standards, several professional objectives and interests from the CST Strategic Plan 2014-2016 are supported and benefit from these activities:

* (1.2) Continue building CA stakeholder participation into IT development projects and encourage consultations and visits with CST Staff
* (2.5) Review and develop means to accelerate technology adoption processes
* (3.1) Lead the implementation of or serve as an early adopter for Federal CIO initiatives.
* (4.4) Build continuous collaboration mechanisms with IRM & DS management.
* (6.2) Create rotational assignments for CST staff
* (6.3) Create a performance matrix and standards for CST staff
* (6.6) Hire more technical talent in key roles throughout CST

By elevating system quality to a more consistent and professional standard worthy of publication, solutions shared within CST and with other collaborative organizations will be presented in a manner easier to understand, to review for compatibility, validation and for use with automated integration tools. As new technology is introduced to the Bureau or offered to other federal adopters for integration, winning over stakeholders by clearly articulating how a system works is critical to its success. System behavior becomes easier to anticipate when internal structure and interfaces are standardized. Murky and obscure system level requirements which business stakeholders are unaware of during early elaborations will be explicitly defined so common issues described in this document are avoided, which may distract them during system review.

Standardization provides training and guidance material for CST staff making it easier to manage. Although this guidance does not necessarily facilitate CST personnel developing code themselves, It does give them insight to recognizing signs of best practices being exercised or the absence thereof.

Modern developers are under pressure of a fast-paced environment where software quality standards are often set aside for another day due to schedule, personnel, or budget constraints. Most developers try to produce good code, but if they are not given guidance to follow coding standards, they will generally fail by increasing the probability of defects with the next developer and simply charge more technical debt to the system.

Based on twenty years of research in software engineering & business IT, CISQ non‐functional requirements lay at the core of the CISQ standards & recommendations. This paper raises the importance for good coding practices at the code unit level and assuring value for the business at the system level if balances of these factors are not thoughtfully addressed. Based on CISQ projections, the relative cost of nonconformance is estimated to aid in prioritization.

# Key Coding Standard Factors

The Consortium for IT Software Quality is co‐sponsored by the Object Management Group and the Software Engineering Institute at Carnegie Mellon University with contributing members from 24 companies, many of whom are Fortune Global 200 Companies. The consortium publishes automated quality measures for technical characteristics as well as underlying rules for good architectural and coding practice. CISQ classifies 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 Level under four CISQ quality characteristic sub classifications defined by ISO/IEC 25010, a series of Software product Quality Requirements and Evaluation standards.

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Good Coding Practices**  **(Unit Level)** | **Good Architectural Practices**  **(Technology/System Levels)** |
| **RELIABILITY – RESILIANCE & DEPENDABILITY** | 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 & VULNERABILITY** | 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, ADAPTIBILITY & CHANGEABILITY** | 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

These coding quality factors are language agnostic, and can be applied to any development, the factors themselves are not tied to a particular languages, but rather follow object oriented design principles. The prime goalis that **consistency and transparency across all application development is the most important guideline in producing quality application code.** These factors will be addressed in detail for each programming language in separate style guides which will detailThe following areas of interest for each programming language for example C#, C++, and JAVA. The subset of quality characteristics described below represent the priorities for the Division with high level approaches to mitigate them and factors to grade systems against.

## Reliability ⎯ Resiliency & Dependability

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

* 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

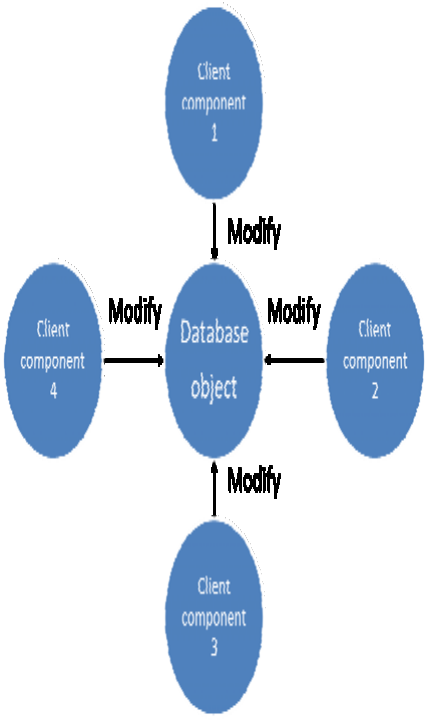
When it comes to checking the consistency of the error management in an application, the challenge is making sure that every exception thrown receives an appropriate treatment down the chain, across software layers, and so on. The software component raising the exception or error might not be able to 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.

Designed thoughtfully, with unit and system levels in mind, error management escalates to operations teams with enough information to improve human response times and mitigation to preempt widespread problems.

* 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, corrupted 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 straw man database access example illustrates 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 well tested code is at the origin of many data corruption cases. System Level analysis to validate data integrity will take into consideration all accesses to databases 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. Alternately, developers should consider using well vetted parallel safe libraries instead of creating their own and proving that they work comprehensively, to cover edge conditions unforeseen by developers.



More insidious cases of data corruption are related to the implicit conversions that occur between two compatible data sources such as the ones found in the different SQL dialects or data types between application and database languages. 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 a non-numeric character is part of the string then the further arithmetic operation will fail resulting in a runtime error. This fault is created due to inappropriate reliance on language features which may save time when writing code but it implies an assumption which may affect data quality when not asserted explicitly.

There are similar but slightly different issues in COBOL 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 J2EE front‐end is moved into a smaller area. However, the data will be definitively truncated leading to an invisible but unfortunate data corruption.

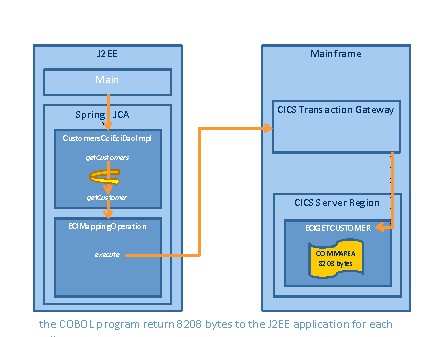
The solution is to use automated code quality tools to enforce detection of data truncation and verify code quality at an early stage of development where potential errors are less costly to correct.

## Performance Efficiency

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

* 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 are executed on a remote server ‐ web service, database, file system. More precisely the post‐mortem analyses of many performance related crashes have shown that the root cause of such failures is ‘buried’ calls to external resources 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 costly SQL access. A concrete example of this anti‐pattern between a Java layer and a mainframe back end is shown in the following figure where a Customer Information Control System (CICS) transaction calls an external resource:



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. While this is COBOL, the same could be applicable to any stored data object. 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 is excessive consumption of CPU cycles leading to an additional cost of running the server.

With system level analysis of the critical code sections, queue theory principles can be applied to pre-stage and prepare execution of code to work asynchronously to best use resources and minimize processing time against a series of long running steps. This of course relies on quality code to prevent data corruption.

* 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 application 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.

## Security - Identifying Vulnerabilities

|  |  |  |
| --- | --- | --- |
| **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 |

* Detecting unsecure dynamic SQL queries thru multiple layers:

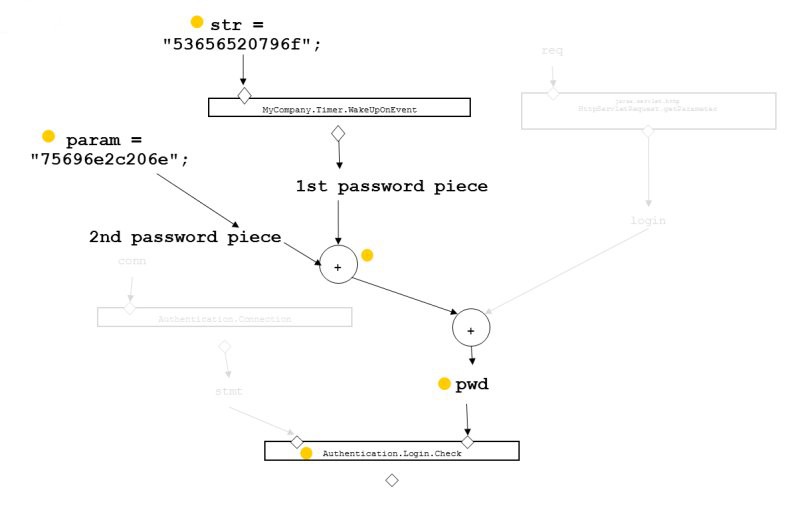
Data intensive enterprise applications may manage data update rights through a single mechanism (layer of classes, SQL stored procedures...). When looking at the code responsible for executing a SQL table update, the difference between good and bad coding isn’t always obvious and requires a higher degree of code comprehension. Good and bad outcomes are possible despite invoking the same calling mechanism below:

ResultSet rs = stmt.executeQuery( myquery )

The more secure version of the update statement will have the string ‘myquery’ built with a call to a predefined stored procedure, while the insecure version of the code contains a dynamic SQL language 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 running the executeQuery call. 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. Where possible code should deny threats avenues of exploit through least privilege principles, improve detection of attempts and to limit damages if ultimately exploited. 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.

* Detecting backdoors:

Use of backdoor approaches relies on secrecy of obscure code for security. When a system is asked to provide credentials to access the authentication server, an elaborate set of application steps may be performed to produce them. . Credentials may be 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. The weakness of this approach is that it relies on code secrecy instead of secrecy of keys which may be rotated and changed when necessary. Changes to code are harder to plan and ultimately is subject to leakage when reviewed.



To detect a backdoor, looking inside a single code unit or technology layer won’t be of any help. Automated tools will have difficulty in discerning whether code instructions are semantically carrying out misguided backdoor steps or is performing legitimate data processing. Once again, a System Level analysis with additional dataflow capabilities to identify strings variables used for authentication is needed.

## Maintainability - Changeability & Adaptability

|  |  |  |
| --- | --- | --- |
| **Coding Errors** | **Context Required** | **Business**  **Impact** |
| Cyclomatic complexity | Unit Level | 25% |
| Hard coding of literals |
| Excessive program 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 |

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.

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 verifiable on a file by 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 expectancy 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 application 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 reasonably modest changes 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 have the potential to block and eventually kill an application or plague its continued use with intermittent faults. The cost of 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. To ensure that a modern multi‐layer application built with a mix of technologies meets its non‐functional and structural requirements, System Level analysis is essential to help prevent architectural degradation which occurs as consequence of continued maintenance.

# References:

# Adopted for division level use from: “How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations”, produced by Consortium for IT Software Quality (CISQ) <http://it-cisq.org/>