



DETECT • PROTECT • ADVANCE

Customer ABC

February, 2011

SOFTWARE: ABC System

QUICK CHECK SERVICE

SCORECARD REPORT

SAMPLE REPORT

Quick Check Service: ScoreCard Report

PSC is the leading provider of independent software quality assessment services. In business since 1993, PSC has provided software inspection services to the majority of the commercial and federal marketplaces. This report contains the results of PSC's proprietary inGenium technology integrated with multiple commercial off the shelf software technologies.

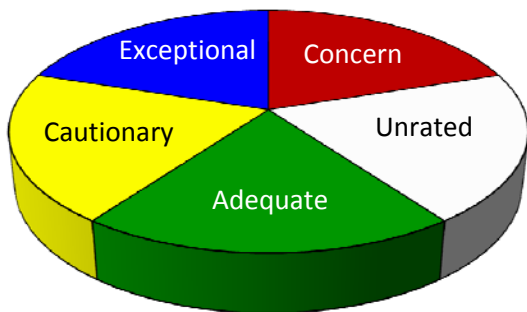
The Quick Check service and report is designed to allow prospects and customers to quickly and easily gain some initial insight into broad areas of software risks at a low cost of entry. The data contained in this report has not gone through PSC's assessment processes and therefore consumers of this information are advised this data may contain high probabilities of false positive results which may affect its true representation of risk to your program.

PSC may provide some interpretive guidance based on its past experience with systems of like size, language and industry to recommend an actionable plan for your company to consider based on the results contained in this Quick Check report.

We appreciate the opportunity to work with your company and we are looking forward to building a successful relationship.

COLOR DEFINITIONS FOR CHARTS

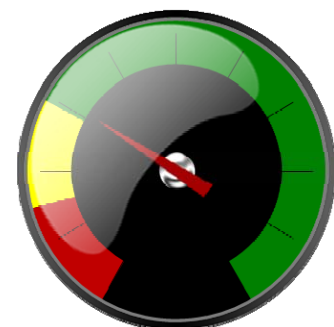
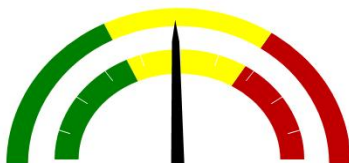
The charts of this report summarize more detailed categories.



Unrated	Indicates Either Insufficient or No Software System Artifacts Were Available for Analysis.
Exceptional	Indicates Process Area Proactively Addresses Quality Risks.
Adequate	Does Not Pose Any Quality Risk to the Software System.
Cautionary	Poses Quality Risk to the Software System and Should be Addressed in the Next Release.
Concern	Poses Significant Quality Risk to the Software System and Must be Addressed Immediately.

COLOR DEFINITIONS FOR GAUGES

The color within the gauges contained in this report represent a statistical comparison of the target software system with "like" systems PSC has processed. The variables driving the comparison are computer language (C, C++, C#, Ada, Java, etc...), system size (measured in lines of code) and industry (financial services, manufacturing, aerospace & defense, communications, etc...).



Average	Does Not Pose Any Quality Risk to the Software System.
1 Sigma Worse than Average	Poses Quality Risk to the Software System and Should be Addressed in the Next Release.
2 Sigma Worse than Average	Poses Significant Quality Risk to the Software System and Must be Addressed Immediately.

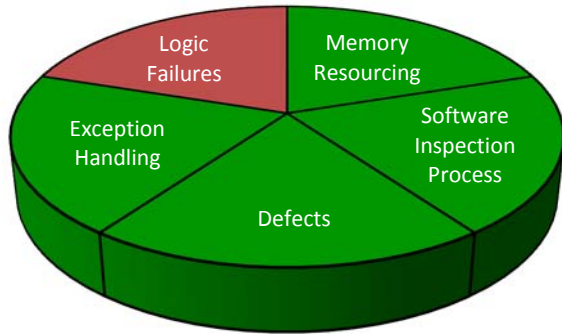
SAMPLE REPORT

C/C++ PIE CHARTS

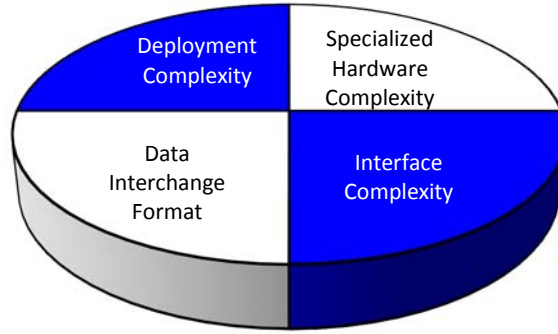


Basic Metrics	
Assessed Lines	43,687
Without Comments	22,222
Files	104
Classes	34
Methods	1,657

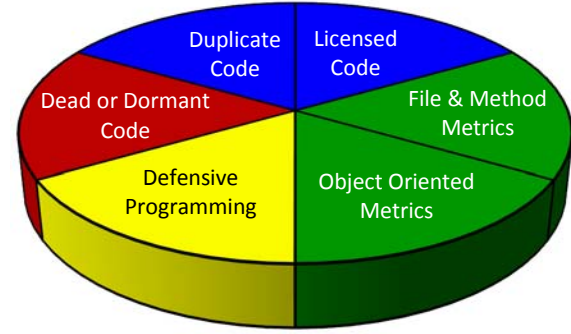
Inspection Attributes



Complexity

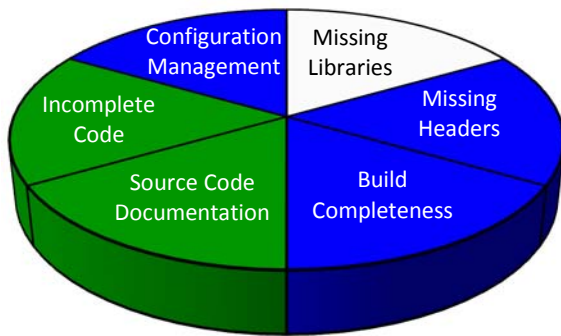


Structural Metrics

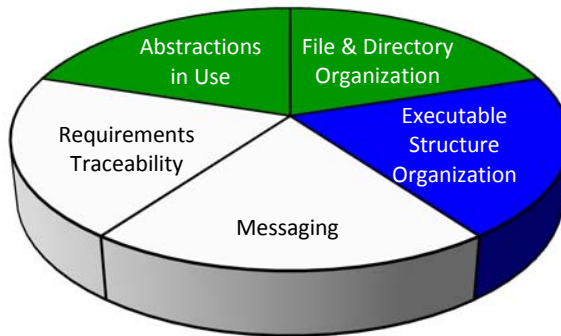


SAMPLE REPORT

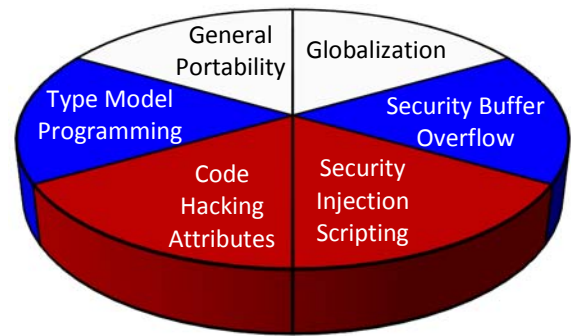
Code Completeness

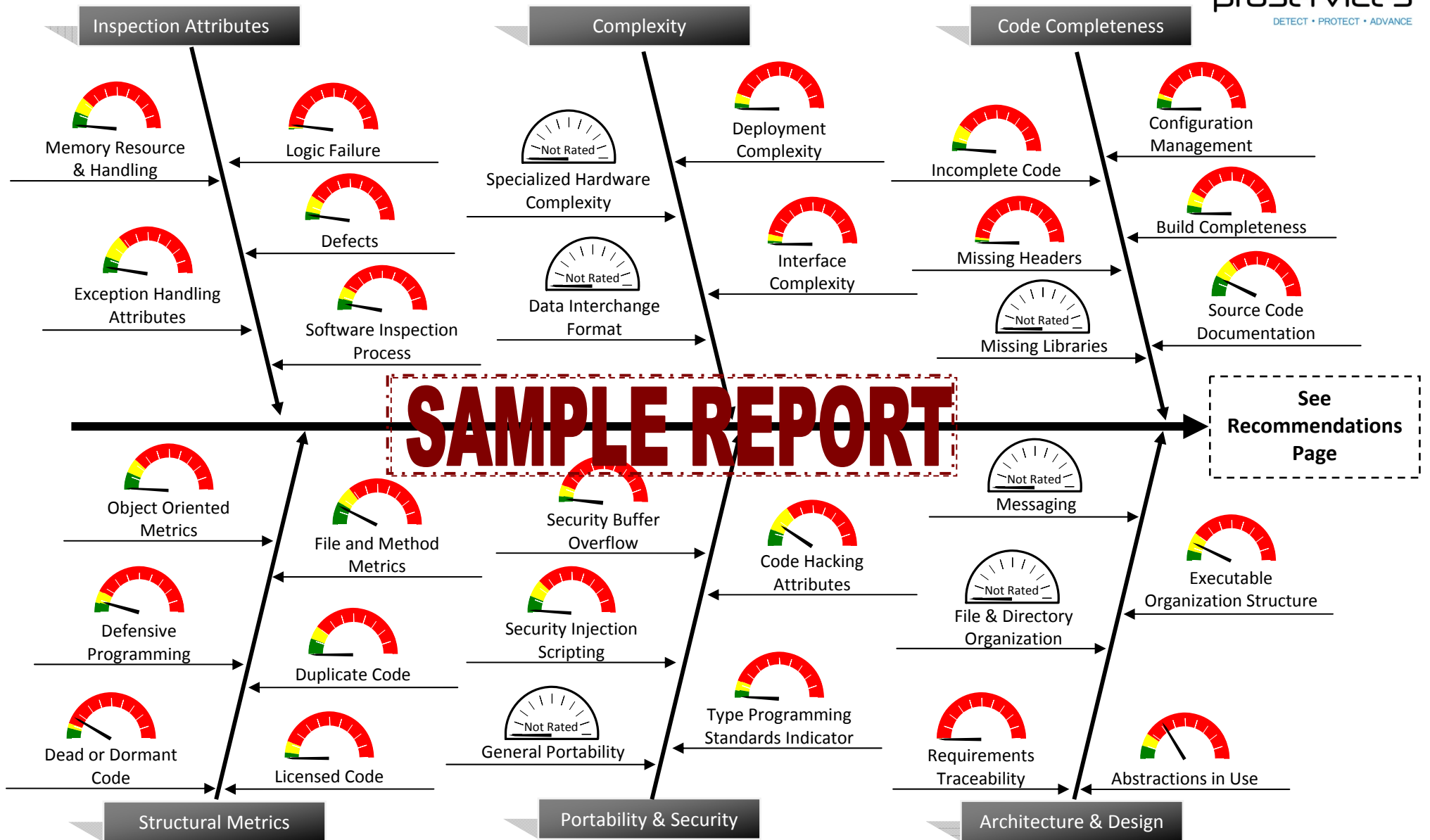


Architecture & Design



Portability & Security



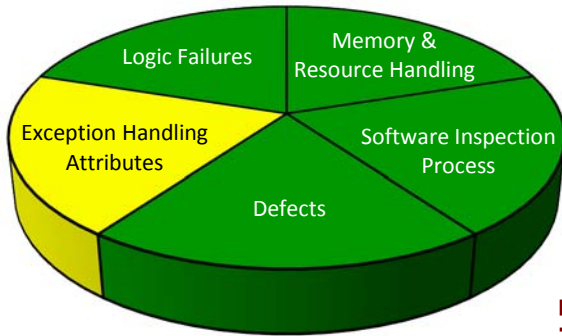


C# PIE CHARTS

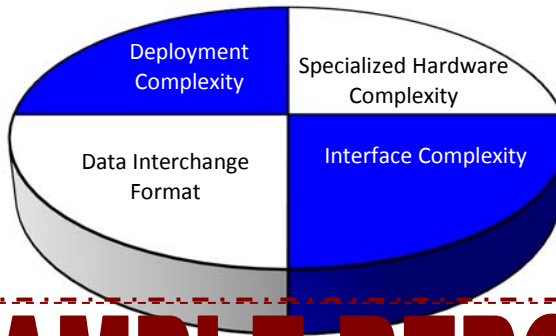
Basic Metrics	
Assessed Lines	807,252
Without Comments	566,401
Files	1,756
Classes	3,761
Methods	51,642



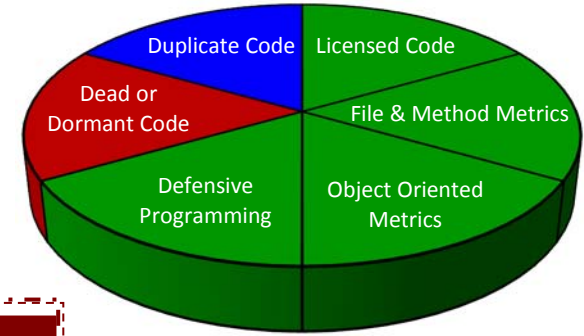
Inspection Attributes



Complexity

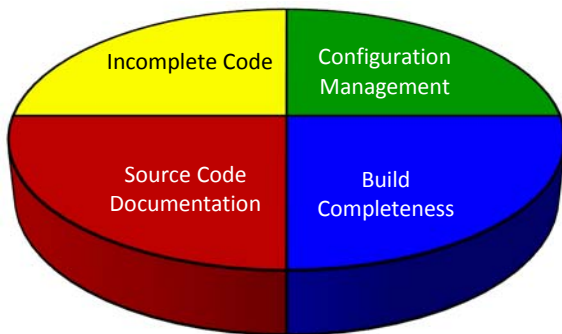


Structural Metrics

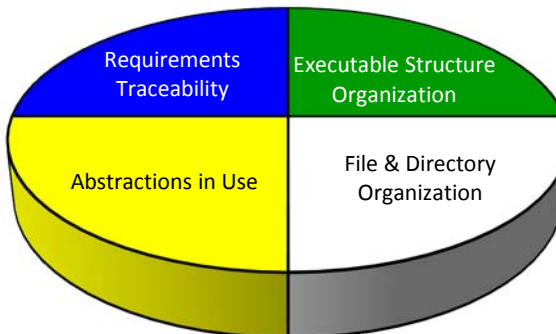


SAMPLE REPORT

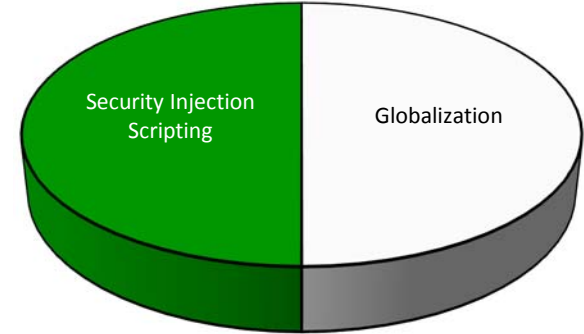
Code Completeness

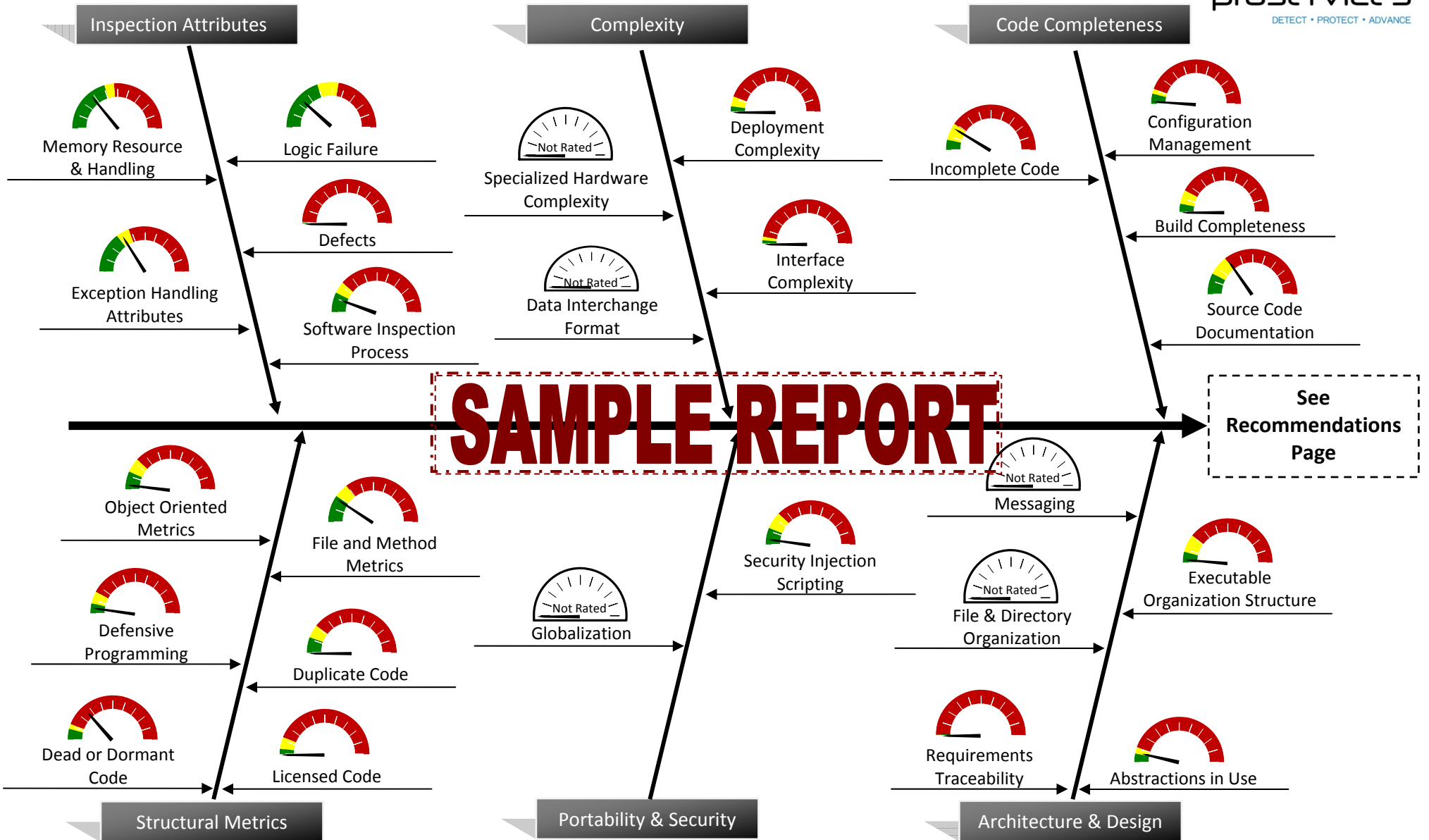


Architecture & Design



Portability & Security



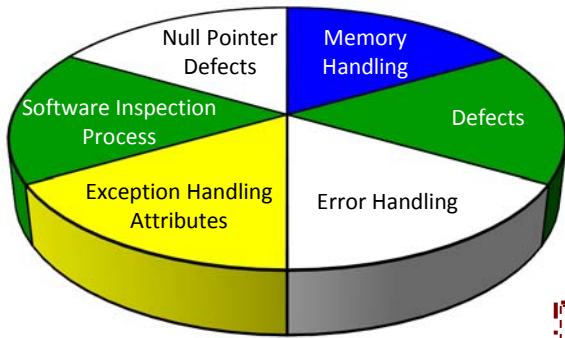


JAVA PIE CHARTS

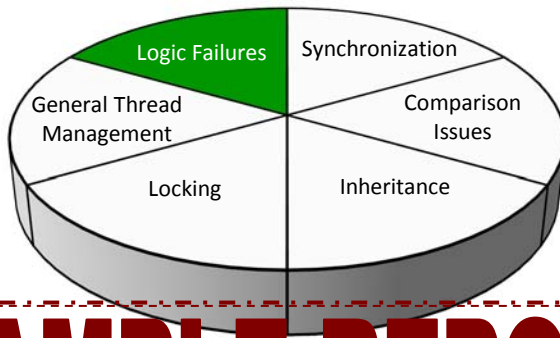
Basic Metrics	
Assessed Lines	74,876
Without Comments	27,981
Files	374
Classes	418
Methods	3,926



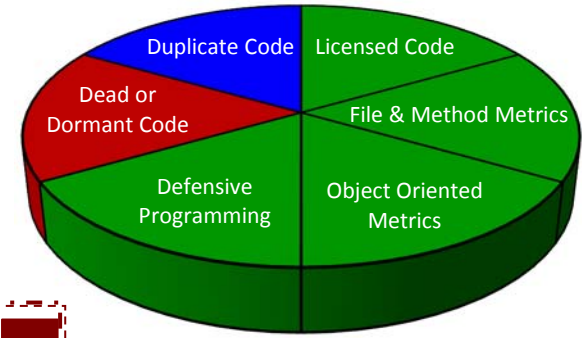
Inspection Attributes



Complexity

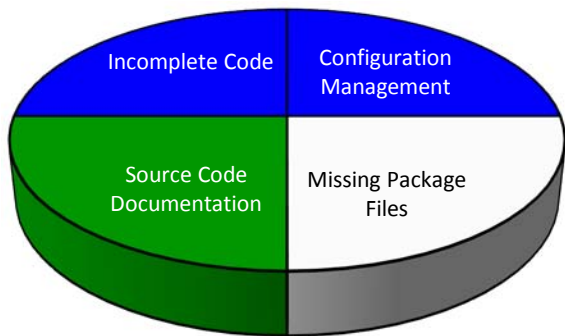


Structural Metrics

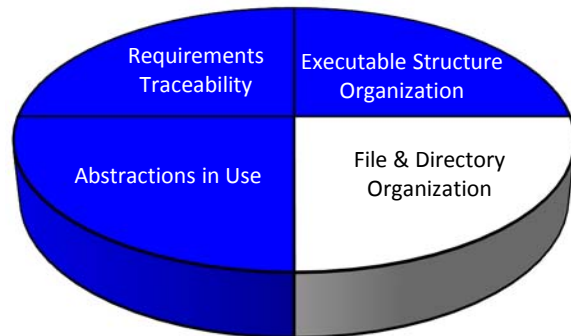


SAMPLE REPORT

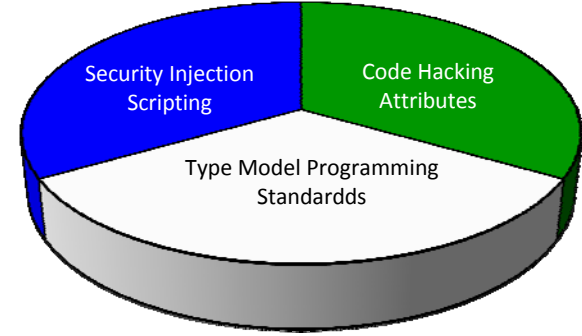
Code Completeness



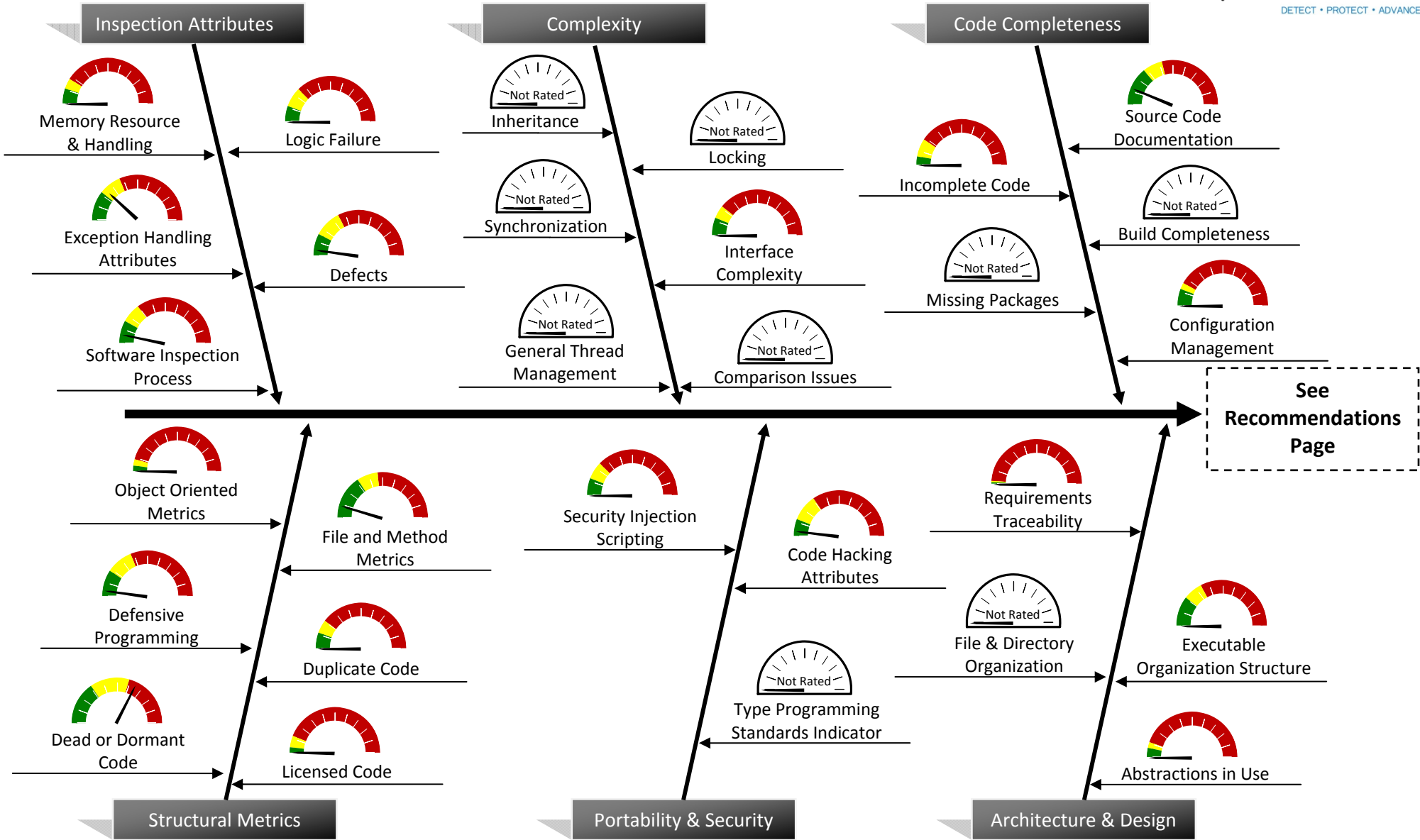
Architecture & Design



Portability & Security

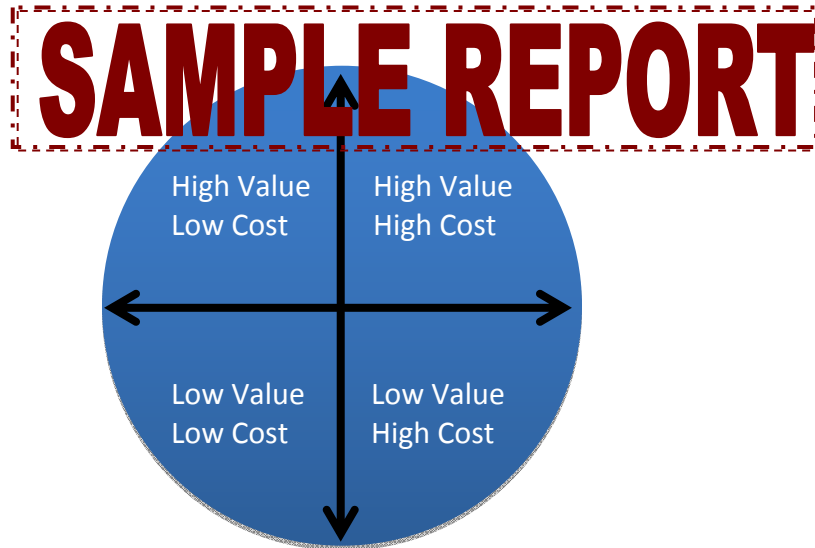


SAMPLE REPORT



Our Recommended Approach

Based on the Quick Check service results, PSC believes the best approach is to combine our **Check-Up** and **Targeted Deep Dive** with emphasis on security, however modify these services to a “Targeted” approach. This service combination offers a broad and at the same time focused analysis in characterizing both the general quality and security of the ABC system. Both of these services are static analysis processes which offers a low risk technique to provide a lot of value without distraction to your development and testing teams. In addition, static techniques are not dependent on the effectiveness of your test beds to identify risks in the code.



Quote for Service Recommendation

Our services are normally quoted using our published price sheet which is based upon the number of lines of code combined with leveraging PSC’s historical yield averages for software systems projects the amount of material for review per service offering. For this opportunity PSC is experimenting with a new sales process and pricing methodology, which consists of performing an initial Quick Check service to understand the topography of the resulting data and provide a quote for a recommended set of services based upon “actual” instances versus using the normal “projected” instances needing review.

Option A – Instances Based Pricing for Check-Up & Targeted Deep Dive services (50%):

- \$xx,xxx

Option B - Instances Based Pricing for Check-Up & Targeted Deep Dive services (30%):

- \$xx,xxx

Glossary of Terms

This section is a summary of each of the detailed pie charts. (Content within each category will vary by language and services)

Architecture & Design - This section reports on the use of architectural elements such as file and directory organization, executable structure organization, messaging, requirements traceability, and abstractions in use. The section is often considered related to the design of architecture but it is focused on description of actual architectural elements and how they may support or impede system goals. This is done by examining architectural artifacts of the design process as found in the source code or other related documents.

Code Completeness - Based on examination of the code, this section indicates the completeness of a software system. Independent ratings are provided for Missing Libraries, Missing Headers, Build Completeness, Source Code Documentation, Incomplete Code, and Configuration Management.

Complexity - Increased complexity is often a function of system requirements and not directly a measure of system quality. However, increased complexity does impact a system and can lead to the necessity to improve other software development processes in order to manage the increased complexity. This section focuses on interface complexity, data format interchange, specialized hardware complexity, and inter-language complexity.

Inspection Attributes - This section contains a review of the manufacturing inspection processes and their attributes.

Structural Metrics - This report section provides quantitative measures of several software characteristics. In general, the lower the quantitative measures are the better. Metrics for files and methods include size, low levels of documentation, and cyclomatic complexity of methods. Dead or duplicated code is identified and characterized. Object oriented metrics measure conformity to accepted software design practices. Licensed code provides counts of files that are freely available for use by government contractors and files whose availability may be restricted by legal considerations. The Defensive Programming section contains counts of deprecated coding practices found to exist in the system.

Portability & Security - Portability risks are grouped as issues of globalization, security buffer overflow, security injection scripting, code hacking attributes, general portability concerns, and type model programming standards.

Architecture & Design

File & Directory Organization - File and Directory Organization reviews the approach to file and folder organization as well as file naming conventions and other aspects of code storage practices.

Executable Structure Organization - Typical older style systems (10 years old+) are commonly comprised of a single major executable, often with a handful of minor helper executables. In the older styles of systems it is not unusual to see on the order of 10-20 executables. By contrast, highly distributed architectures tend to have multiple executables in place of the old-style single major executable.

Messaging - This scale identifies the degree to which source code contains evidence of messaging elements such as message formats, queuing, or piping structures and other message processing logic.

Requirements Traceability - This scale identifies the degree to which requirements traceability is evident within the source code. Knowing which part of the code meets which requirements is helpful in planning and executing reuse, porting, or maintenance activities.

Abstractions In Use - This section does not look at abstraction in its object-oriented sense. As used here, abstraction refers to recognizing common software issues and resolving them in a uniform, controlled way with a minimum of duplicated, irregular, or costly code. From this perspective, abstraction includes diverse practices such as providing common services and libraries, using templates, whether standard or custom, and employing design patterns.

Code Completeness

Missing Libraries - This scale indicates the extent to which source code has missing build components. This service is based on the build description contained in the SVD and/or build log (if available). Items identified as missing are typically large groups of Commercial Off-The-Shelf (COTS), Government Off-The-Shelf (GOTS), Operating System (OS), compiler, or other files used by the application but external to it. These files must be made available before the system can be built completely.

Missing Headers - This scale indicates the extent to which the source code contains references to header files that are not included with the source code or not part of the standard enterprise development environment. The absence of such files will prevent the as-delivered system from building if the need for the files is not recognized and if the files cannot be provided from other sources. In this section, the missing files are likely to be individual files missed by lapses in configuration management and/or the build process. The header files may have been kept in libraries separate from the main development libraries. NOTE: Platform and compiler specific files as well as files created by Interface Definition Language (IDL) to C++ generators are listed as missing header files.

Build Completeness - This scale examines the effort required to compile the as-delivered source code and produce one or more executables. This assessment category focuses on the developer supplied build command file (e.g., makefile) and examines the link commands, identifying all build elements to be linked into the final executable.

Source Code Documentation - This scale indicates the degree to which standard prologs are complete, commented out code is present compared to actual documentation, and comment density meets customer standards. Some of this data will be derived from the Structural Metrics section. This information may provide a useful data point when attempting to schedule efforts for reuse, maintenance, or porting of this system. Code may have an apparently good ratio of comment lines to code lines without containing much useful documentation if large amounts of code have commented out or comment lines have been used primarily and excessively to provide white space or character based graphics.

Incomplete Code - This scale identifies the degree to which source code contains incomplete code or data structures. This section focuses on missing functionality as indicated by TBD (To Be Done or Determined) or To Do comments in the code. This category of missing code is defined as incomplete code that is not sufficiently integrated with the system to cause logic failures. Logic failures due to stubbed functions, empty statement bodies, incomplete data structures, and related issues are treated under Inspection Attributes.

Configuration Management - This analysis attempts to identify, based on artifacts received, the type, nature, and completeness of development configuration management procedures utilized with the code under review. In some cases, we can observe the presence of source code comments indicating the type of system used, and the comments will give some indication of completeness. In many newer cases however no such documentation comments will be present and no such analysis can be performed. Additionally we can make observations of completeness based on the degree of Missing Headers section contents.

Complexity

Deployment Complexity - This section provides an analysis of the product deployment, computational elements, and element interdependencies.

Data Interchange Format - This scale identifies the degree to which the source code performs complex data formatting or data conversions. Gateways between networks with different protocols are excellent examples where intensive format interchange may be required. System changes or porting requirements may turn a carefully tuned application into one that will fail to meet performance requirements. Because this type of processing is low visibility infrastructure with highly complex details understood only by a few specialists, its role within a system as a whole can suffer. There may be simple errors in formatting due to lack of expertise in one of several protocols in use. Sequential processing of aggregated data structures may be disrupted by programmers who are unaware of the system as a whole.

Interface Complexity - This scale identifies the degree to which the source code's complexity is increased by complications in the interface layers, violations of interface standards, complications in managing interfaces in the source code.

Specialized Hardware Complexity - This section discusses the use of specialized computing devices such as DSP chips, FPGAs, or ASICs, which may have required the introduction of languages and/or compilers specific to the hardware. These notations are not to be interpreted as defects, and instances of their occurrence are listed as Informational.

Inspection Attributes

Software Inspection Process - This scale identifies the degree to which the source code reflects software process and Capability Maturity Model® Integration compliance, (CMMI®), specifically as the process relates to coding standards. All evaluations of CMMI software process have been performed based on using the artifacts supplied for review during this service. This capability is currently under construction and will be available in future product assessments.

Defects - This section identifies defects other than those directly addressed by specific topics (e.g., Logic Errors, Exception Handling, etc.) that have been identified during analysis.

Exception Handling Attributes - This scale identifies the degree to which source code contains logic appropriate for error checking and recovery. Certain specific defensive programming practices can reduce the likelihood of errors. Their presence or absence is noted here. Lack of appropriate error handling can make recovery from system problems more complicated and expensive than it has to be.

Logic Failures - This scale identifies the degree to which the source code has indications of incorrect logic structures. These may be caused by undetected typographical errors, by not initializing variables, by misunderstanding of the programming language, or a host of other factors. The net effect is that the system will not perform as desired although the scope of the problem may vary with the type of defect.

Memory & Resource Handling - This scale identifies the presence of standards when managing resources including their acquisition, synchronization, and release. Mismanagement of system resources can slow, stop, or crash a system.

Structural Metrics

File and Method Metric - This scale contains metrics on file and method size as well as method complexity. Also included are counts of methods with more than one return point, methods with more than five parameters, and larger methods that are relatively un-documented. Some average and extreme figures are provided to illustrate ranges found in the metrics values. The overall rating for the File Metrics section is based on the weighted average of individual metrics given by the tables below. The weights applied are based on the impact of measured problems on software quality.

Object Oriented Metrics - Object oriented metrics measure software characteristics as affected by features such as inheritance, abstraction, and polymorphism. Inheritance may have the most significant impact since it increases complexity as measured by concepts such as depth of inheritance. Classes can be related to each other in new ways. The number of class members increases with depth in the inheritance tree. The way in which class methods use data members can yield measures of class cohesion. Multiple direct base classes (multiple inheritance) is highly discouraged and may cause system crashes due to irresolvable naming conflicts. Problems in this area may indicate a lack of understanding of the object-oriented paradigm leading to a more complex and non-standard system. The overall rating for the Object Oriented Metrics section is based on the weighted average of individual metrics given by the tables below. The weights applied are based on the impact of measured problems on software quality.

Defensive Programming - Indication of insufficient defensive programming occurs when either inadequate error checking is done or when there are conditions that can easily lead to defects. Defensive programming indicators are important since they frequently mitigate a potential increase in the future maintenance cost of a system. If error checking is not performed, categories of errors can creep into code undetected, leading to potentially expensive defect detection in the field. In defensive programming, the primary focus is on the quantity of indicators found in the code, versus a review of individual instances.

The counts in all of these categories indicate a lack of defensive programming. The reduction of the counts within these categories can be accomplished with the adoption and implementation of coding standards.

Dead or Dormant Code - This scale reflects the presence within the source code of:

- unused objects
- unused methods
- code unreachable due to preprocessor statements
- code unreachable due to values of control variables
- large portions of code hidden by comment characters.

Some apparently dead code may represent real world functionality that simply is not currently used but may be needed in different circumstances. Frequently the dead code indicated in this report is not accurate as the system may be partial, and thus the methods can be called from outside the supplied software; it can be a library with only a subset of methods used in the current system, etc. Knowledgeable review is suggested before removing dead code from a system. Having dead code scattered through a system complicates maintenance and reuse while adding no value. It also interferes with accurate planning if project size is misrepresented.

The definition for total objects given below makes no allowance for declared data types. This makes the figure provided for percent of all objects that are unused actually come out low since, if known, the number of data types could also be subtracted from the total number of declarative statements.

Unreachable lines and inactive lines are very similar. In essence, inactive statements are those rendered dormant by preprocessor code, and unreachable statements are those created by means of the logic flow within a high or mid level programming language. Inactive lines do not execute because they are in the false path of preprocessor statements like #if or #ifndef. Inactive lines are only found in C/C++ applications. Lines occurring in a file after a return statement are one example of code made unreachable by a high level language statement. Some review of the unused code is required before any other action or conclusions are reached as a number of code approaches can cause code to be indicated as unused when it is actually still in use (e.g. library code etc.) The overall rating for the Dead or Dormant Code section is based on the weighted average of individual metrics given by the tables below. The weights applied are based on the impact of measured problems on software quality.

Duplicate Code - This scale presents the results of a search for:

- Files with programming language statements that match exactly.
- Files with one portion that match exactly in each file while one file contains more code than the other (Semantic Differences – Additive Lines)
- Files that are exact matches except for the presence of semantically irrelevant changes

Licensed Code - The boundaries between a customer's use of software and rights retained by the developer of the software are considered here. Ownership and licensing issues like file use restrictions, vendor name, license type, import-export issues, and liability restraints fall into this category.

Portability & Security

Globalization - Globalization provides an analysis of code that has indications of being difficult to support ports into foreign language domains. Aspects of code such as use of hard coded strings, direct character offset references that would not permit multi-byte character support are examples of the kinds of items searched for as part of this process.

Security Buffer Overflow - The rating in this category is based on identification of buffer overflow potential in the source code. Buffer overflows may present opportunities for attackers to take control and exploit the software system

Security Injection Scripting - The rating in this category is based on identification of injection sources in the code. Interfaces to databases and the operating system are potential vulnerable points for attackers to inject unwanted commands into a system.

Code Hacking Attributes - This scale identifies the degree to which source code contains egregious programming practices, subterfuge or other inappropriate source code structures. By definition, these items serve a purpose separate from the intended purpose of the software system.

Type Model Programming Standards Indicators - The use, abuse, or absence of strong data typing principles shows what kind of type model has been implemented in the application. It is frequently the case in older 'C' code that the type model is nearly non-existent. This condition may or may not be applicable in C++ or more strongly typed languages. Blurring of types is often an implicitly made choice, made during construction of code in contrast to being a design decision.

General Portability - This scale reflects the degree to which the code contains coding practices that are generally recognized to detract from portability but are not specifically addressed elsewhere in the Security and Portability section.