

Assessing Code Quality and Security

Alan Cohn

Mena Kostial

Jay Marchetti

Nicholas Reimer

Michael Riley

David Svoboda

V1.60

Oct 2022

Copyright 2022 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

GOVERNMENT PURPOSE RIGHTS – Technical Data

Contract No.: FA8702-15-D-0002

Contractor Name: Carnegie Mellon University

Contractor Address: 4500 Fifth Avenue, Pittsburgh, PA 15213

The Government's rights to use, modify, reproduce, release, perform, display, or disclose these technical data are restricted by paragraph (b)(2) of the Rights in Technical Data—Noncommercial Items clause contained in the above identified contract. Any reproduction of technical data or portions thereof marked with this legend must also reproduce the markings.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM22-0906

Assessing Code Quality and Security

Overview

Audience, Objectives, and Topics

Mike Riley

Assessing Code Quality and Security – Overview

Let's begin with a brief look at whom this course is intended to benefit and what it's about...

Intended Audience:

This course is targeted to program office or contractor personnel responsible for development, testing, quality assurance, and project management of software-intensive systems.

Welcome!

Assessing Code Quality and Security – Overview²

Objectives:

In this course you will be introduced to methods developed and utilized by SEI to assess the quality and security of mission-critical code implemented in C/C++. At the completion of the course, learners will be versed in SEI's Code Risk Estimation Worksheet (CREW) and the code assessment process for utilizing it. Learners will also gain awareness of SEI's Source Code Analysis Lab (SCALE) tool and process, as well as the associated CERT secure code standards.

Primary Topics Include:

- Software Quality
- Code Analysis
- The Code Risk Estimation Worksheet (CREW) and the CREW Assessment Process
- The Source Code Analysis Lab (SCALE) Process and Tool for Security Analysis

Agenda

Assessing Code Quality and Security

- 4-hour (total) course
- Broken into three 75-minute parts
- 5 to 10-minute break between parts
- Optional Q&A time at the end



Part	Duration (minutes)	Topics	Discussing
1	10 15	Introduction Software Quality	• Course Introduction and Overview • Quality Attributes • Code and Quality Standards
	50	Code Analysis	• Static and Dynamic Analysis • Static Analysis Tools • Structural Metrics
2	75	Code Risk Estimation Worksheet (CREW)	• Introduction, Overview, and Objectives • Inputs, Process Flow, and Outputs • Code Quality Aspects • Analyst Questions and Scoring Risk • Supporting Analytics (Scripts)
3	40 20 15	CREW SCALe Conclusion	• Tab Walkthrough • Overview of SCALe Security Analysis • Briefing CREW Results • Recommendations for Acing CREW • Potential Future CREW Improvements

Assessing Code Quality and Security

Discerning Software Quality via Code Analysis

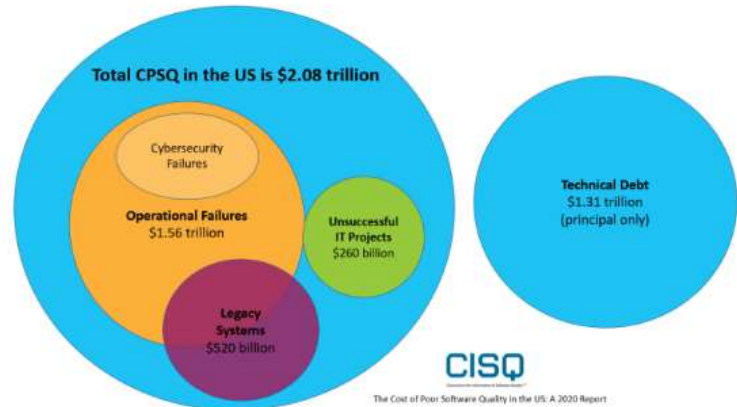
Quality Attributes

Jay Marchetti

The Risks of Poor Software Quality

- Particularly for mission- and safety-critical systems, poor code quality entails manifold risks. For example, consider the risk that a code base is:

- Hard to test thoroughly
- Prone to critical failures / crashes
- Unmaintainable over its expected lifecycle
- Tough to extend for new capabilities
- Exploitable to cyber attacks
- Difficult to harvest for reuse
- Plagued with a multitude of latent defects
- Hard to change without adding new defects



From the report [The Cost of Poor Software Quality in the US: A 2020 Report](#) published in January by CISQ. The report estimates that poor software quality cost the United States economy over \$2 trillion in 2020 due to operational software failures, poor quality legacy systems, and unsuccessful projects. Compared to the country's projected Gross Domestic Product (GDP) of \$20.66 trillion, or the \$1.4 trillion spent on employing IT/software professionals in 2020, it represents a staggering amount of wasted resources.

Copy right © CISQ

We can **estimate** many of these types of risk via code analysis.

Code Analysis

What do we mean by **code analysis** in this course? How does it relate to **code quality**?

- **Code analysis** is the semi-automated analysis of software to assess the degree to which it embodies one or more desired **quality attributes**.
- A quality attribute (QA) is “a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders” [2].
- Certain QAs will be of high importance for a given system. For example:
 - performance
 - maintainability
 - safety
 - interoperability
 - security
 - testability
- While others may not be important for the particular system at hand, e.g.:
 - portability
 - scalability

“Today’s code quality issue is tomorrow’s zero day.”

Code Analysis – Greenfield and Brownfield

Can we really determine the QA's from both **in-development** and **existing** (or even legacy) code?

Architecturally significant Quality Attributes (QAs) drive the system's architecture.

- But QAs can also be viewed as **aggregates of code characteristics** that can be tested for at runtime or identified within the source code.

However, most QAs are difficult and expensive to add later, so...

- Best practice: Apply **code analysis** early on and verify that the essential QAs are designed into the system's "ground truth" artifact: the code.

- ★ Use code analysis early in the software development life cycle to verify essential QA's.
- ★ But code analysis can also be done "after the fact" to assess whether a code base exhibits the desired (or *required*) QA's. This is a common code analysis use-case...

Quality Attribute - Examples

Accuracy	Flexibility	Performance	Safety
Adaptability	Integrity	Portability	Scalability
Availability	Interoperability	Readability	Security
Correctness	Maintainability	Reliability	Testability
Deployability	Modifiability	Reusability	Usability
Efficiency	Modularity	Robustness	Understandability

Q: What makes Security **special** among the many possible QA's?

A: It's the only one focused on thwarting an intelligent and motivated adversary...

Assessing Code Quality and Security

Discerning Software Quality via Code Analysis Standards

Jay Marchetti

Coding Standards

A **coding standard** provides a set of coding practices for use by the development team to enhance source code quality.

There are two (2) main types of coding standards:

Coding Standard Type	Intent	Usage
Stylistic	Project and / or organizational conventions for consistent, readable, and maintainable code.	Most modern organizations have adopted stylistic guidelines for all new code, which their developers are strongly encouraged (or required) to use.
Critical	Language restriction rules for safety-critical, mission-critical, and/or security-critical code.	Rule-based standards are important for systems where security or criticality are key, but they demand additional investment in developer training and tools.

- Stylistic coding standards are a best practice for all software development.
- However, additional coding standards are essential for critical systems.

Stylistic Coding Standards

Stylistic coding standards – good ones focus on two key concerns:

- Reducing the likelihood of coding errors.
- Enhancing the maintainability of the code over its entire intended lifetime.

Stylistic coding standards – are often language-specific, but will generally cover:

- White space, indentation, alignment, and module layout
- Brace (block delimiter) placement
- Naming conventions for modules, classes, functions, variables, and constants
- Commenting module headers, file sections, classes, functions, code blocks, etc.

- Most organizations serious about software quality utilize a stylistic coding standard.
- Good style implies that you “Write code for people first, computers second” [1].

Why are Stylistic Coding Standards Important?

As a new hire, you're assigned to maintain one of your company's two software products. Each is 2.5 MSLOC of C++ code. Below are code snippets typical of each:

Product A

```
1 void main(int argc, char* argv[]) {
2     while (1){ int in;bool cit[10];
3         cin>>in; cit[0]=0;
4         for (int i =1;i<in; ++i) cit[i]= 1;
5         if (!in) exit(-1) ;int m=1; int cp=0;
6         while(oneleft(cit,in)!=12)
7         { if(oneleft(cit,in)==-2){
8             ++m; cp=0;cit[0]=0;
9             for (int i=1;i<in;i++) cit[i]=1; }
10        add(cit,in,cp,m);}
11    cout<<m<<endl; }
```

Ugly Code!

Product B

```
1 // Main - Start and initialize the system
2 void main(int argc, char* argv[]) {
3     bool bDone = false;
4     bool bMotorInitialized = false;
5     while (bDone == false)
6     {
7         // Start watchdog timer
8         int nRetVal = nInitWDog(WDOG_PERIOD_MSEC);
9         if (nRetVal != NO_ERROR)
10        {
11            cout << "Watchdog init failed with ID = " << nRetVal << endl;
12        }
13        // Loop processing incoming commands until STOP issued
14        int nCommand = nReadCmdChannel();
15        if (nCommand == STOP)
16        {
17            bDone = true;
18        }
19        else
```

Clean Code!

Your assignment entails adding new features and fixing reported bugs.
Question: On which product would you likely be most productive?

Answer: B

Critical System Coding Standards

A **critical system** is a software-intensive system that is safety-critical, mission-critical, and / or security-critical.

A critical coding standard should be required for all systems deemed critical, but organizations must realize that this will drive:

- Additional effort, tools, and training for the system development. Hence, identify and address these needs **early** in the project.
- The need for static analysis (SA) tools that support checking code against critical coding standards, which is **essential for critical systems**.

- Determine the security-, mission-, and / or safety-criticality of your system early.
- Using SA tools to check code against criticality standards is essential.

Coding Standards for Critical Systems

Coding standards for **Critical Systems** are rule sets that:

- Restrict the use of some features of a programming language – e.g. malloc(), goto, etc.
- Limit complexity and the amount of code within modules – e.g. Cyclomatic & SLOC
- Disallow dead code, e.g. unreachable, no effect, or commented out code
- Specify how to use expressions, the pre-processor, portable data types, etc.
- Restrict the use of SOUP (Software of Unknown Provenance)
- State which rules can be deviated from and how to manage rule deviation
- May overlap with stylistic standards, i.e. on commenting, naming conventions, and brace usage

Our focus in this course is on mission-, safety-, and/or security-critical systems.

Coding Standards for Critical Systems²

Criticality Standards help organizations avoid creating high-risk critical system code.

- Software Quality
 - ISO/IEC 9126 and 25010 – Software Engineering Product Quality
 - Hersteller Initiative Software (HIS) – European Automotive Consortium
- Software Safety
 - DO-178C – Software Considerations in Airborne Systems & Equipment Certification
 - USAF 91-119 – Safety Design for Nuclear Weapon Systems Software
- Software Security
 - SEI CERT Secure Coding Standards for C, C++, and Java
 - MITRE CWE – Common Weakness Enumeration

Examples

But these standards can also provide a basis for characterizing risks within existing code.

Mission- and Safety-Critical Coding Standards - Examples

MISRA: Motor Industry Software Reliability Association for C and C++

- A well-known consortium standard first created by the automotive industry in 1998
- Widely accepted in many sectors, including medical, defense, and aerospace
- A “draconian” rule set that forms the basis of several other critical code standards

AUTOSAR: Guidelines for Use of the C++14 Language in Critical & Safety-related Systems

- Open standard w/ 342 rules (many from MISRA C++:2008) for safety-critical C++ code

JSF AV++: Joint Strike Fighter Air Vehicle C++, Lockheed Martin

- Developed for the JSF by Bjarne Stroustrup, creator of the C++ language
- Based on MISRA C++ but openly available; widely accepted for safety-critical C++ code

HIC++: High-Integrity C++, PRQA (now Perforce Software)

- Open standard w/ 155 rules emphasizing “maintenance, portability, readability, and robustness”

JPL Institutional Coding Standard for the C Programming Language

- Based on MISRA C and NASA “Power of 10”
- Covers multi-threaded, safety-critical C code

Software Quality Standards - ISO/IEC 9126 and 25010

Taxonomies of Software Quality – Functional and Structural

Zooming out from coding standards, there exist several **Software Quality Standards**.

Two well-known software quality standards are ISO-9126 and, more recently, ISO-25010.

- These standards cover both **Functional Quality** (external, i.e. requirements) and **Structural Quality** (internal, i.e. construction).

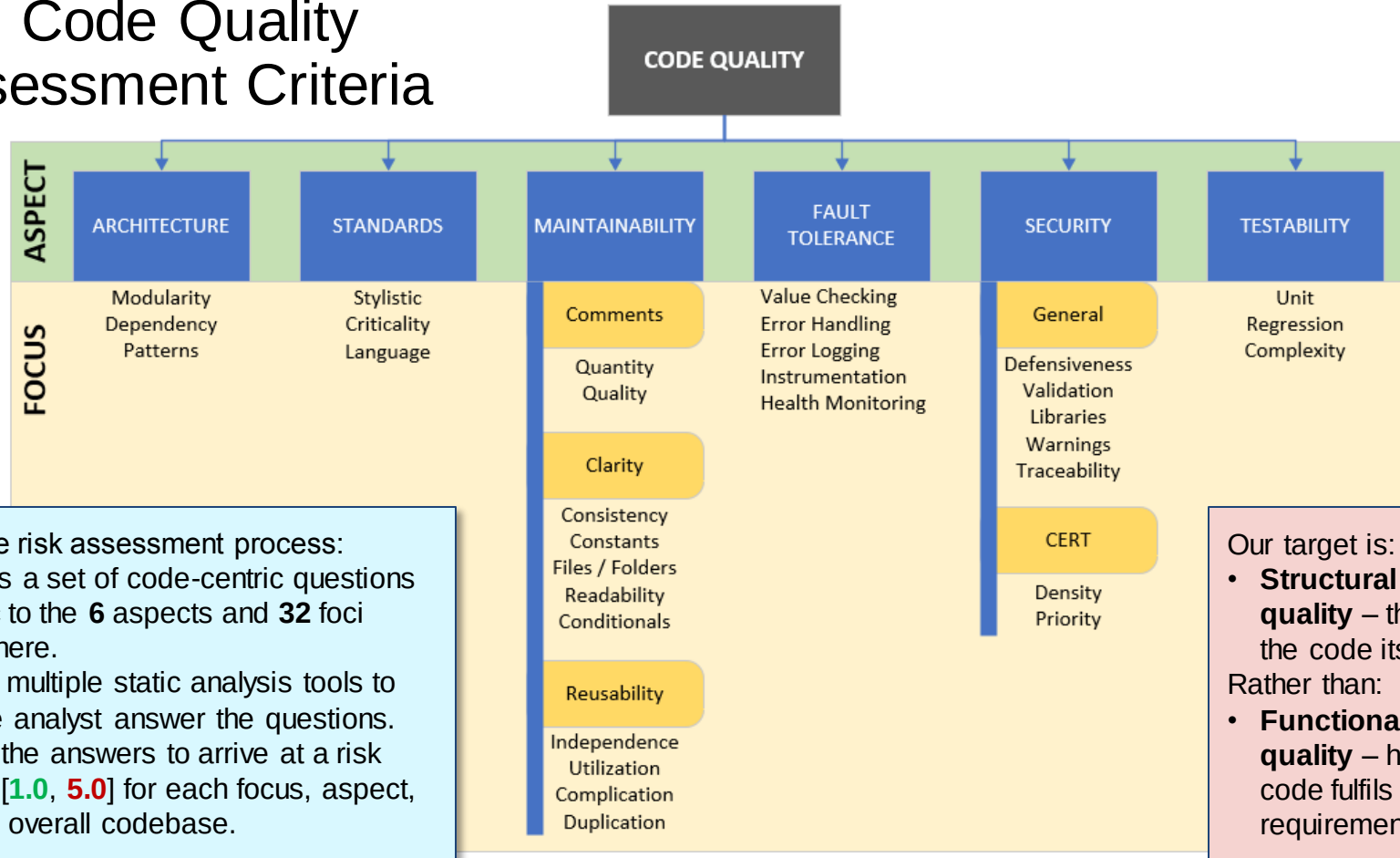


But what if all you have is the source code, w/o requirements or other design documentation?

SEI Code Quality Assessment Criteria

6

32



SEI's code risk assessment process:

- Employs a set of code-centric questions specific to the **6** aspects and **32** foci shown here.
- Applies multiple static analysis tools to help the analyst answer the questions.
- Scores the answers to arrive at a risk level in [1.0, 5.0] for each focus, aspect, and the overall codebase.

Our target is:

- **Structural code quality** – the quality of the code itself.
- Rather than:
- **Functional code quality** – how well the code fulfils mission requirements.

Assessing Code Quality and Security

Code Analysis

Static and Dynamic Analysis

Jay Marchetti

Static Analysis

Static Analysis (SA) is the analysis of software while it is not executing.

While SA may be done on Java or .NET bytecode or on binaries, typically to address security concerns about the software supply chain...

- Most often SA implies **analysis of source code**. This is the meaning of SA for this course.

Furthermore, while peer reviews may be considered a form of SA...

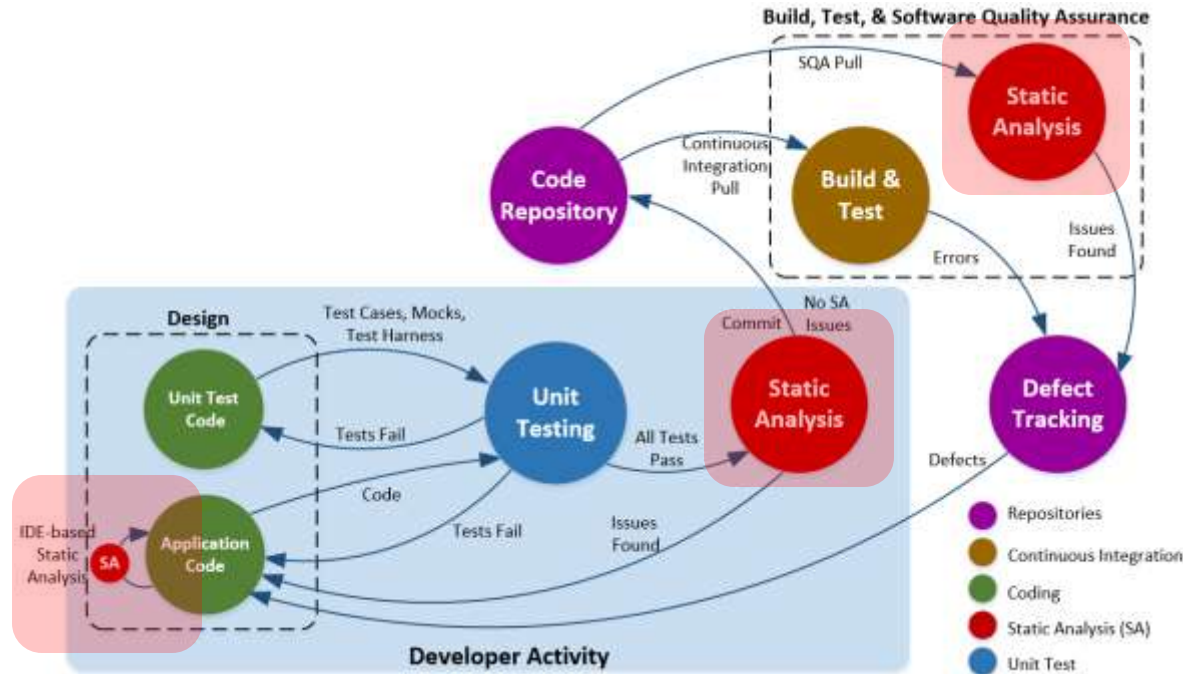
- SA more typically means **automated or semi-automated source code analysis using tools**. Again, this is the meaning of SA for this course.

- Static Analysis (SA) most commonly means analysis of source code.
- SA most commonly is automated by using one or more SA tools.

Static Analysis²



Best Practice: Embedding SA within the Software Development workflow.

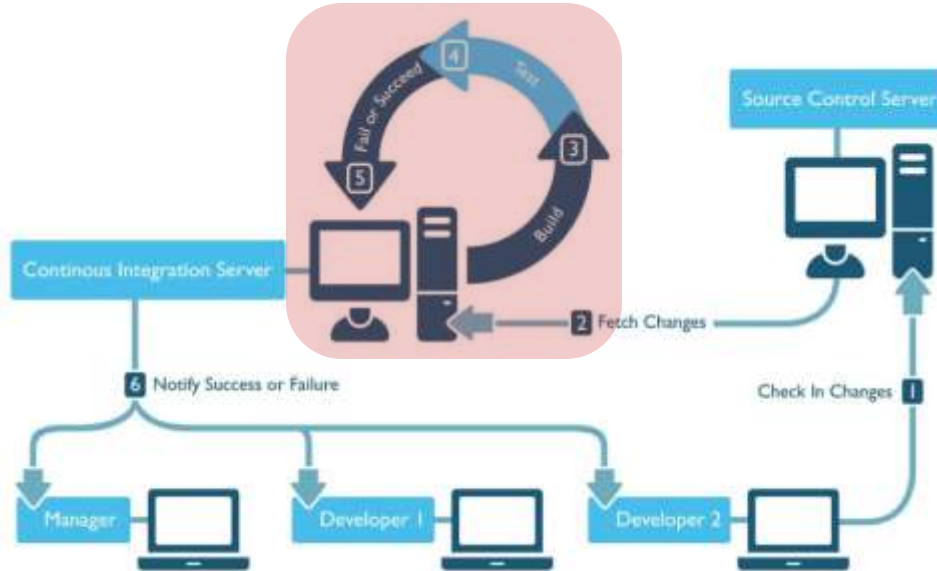


- Static analyses is done early and throughout the Software Development process.
- Ideally, three different SA tools are used here.

Static Analysis³



Best Practice: Using SA as part of a Continuous Integration (CI) system.



- CI continually merges, builds, and tests a software system when new code commits are detected. [4]
- CI encourages an iterative, “fail fast,” Agile development model.
- CI is an inherent part of DevOps.
- SA can be run on the CI server along with unit test and regression test suites.

Dynamic Analysis

Dynamic analysis (DA) is the analysis of software while it is executing.

Dynamic Analysis validates executable software via two primary means:

Testing

Monitoring

- **Testing** seeks to reveal latent design and coding defects:
 - As early in the development process as possible, e.g. Unit Testing.
 - Software testing covers unit, integration, system, regression, and acceptance testing.
 - **Monitoring** utilizes field & production inputs rather than tester-selected inputs to:
 - Validate performance, system health, security, licensing, and end-user utilization.
- Dynamic Analysis (DA) encompasses software testing and run-time monitoring.
 - DA and SA are complimentary – both are essential for creating high-quality code.

Software Testing – Types and Levels

Software testing applies input data, signals, and events in order to reveal defects.

Testing Type	White Box	Black Box
Goal	Verify code & design implementation	Validate requirements

Testing Level	Type	Purpose
Unit tests	W	Test an individual unit (function, class, or module) in isolation, using test cases that verify its correct behavior.
Integration tests	W	Group modules and subsystems together to verify interfaces and correct combined behavior.
System tests	B	Evaluate an entire integrated system against specified system requirements.
Acceptance tests	B	Evaluate the system's compliance with business requirements and assess whether it is acceptable for delivery.

Software testing applies V&V through increasing levels of integration and completion.

Unit Testing – Fundamental Code Validation

Unit Tests (UT) validate the correct functioning of a small portion (unit) of code in isolation from the larger system.

UTs are typically created by the developer in conjunction with the unit of code itself.

- UTs are sets of test cases with assertions validating the output for known test inputs.
- UTs often form the basis of regression test suites to validate new system builds.

Unit Testing is **fundamental** to high-quality and maintainable code.

- Skipping or minimizing UT in order to get code into integration testing is a common but costly mistake.

- UT is key to code quality and maintenance. Without it you're developing legacy code.
- Best practice: Code passes UT before VCS commit, and UTs are committed to VCS also.

Assessing Code Quality and Security

Code Analysis

Static Analysis Tools

Alan Cohn

Static Analysis: Considerations for Choosing SA Tools

Consideration	Scope
License pricing	Free to ~\$50K per project
Languages supported	C, C++, C#, Java, Python, Ada, other
Coding standards supported	MISRA, JSF AV++, HIC++, CERT Secure Coding, other
Intended usage environment	DevOps/CI, GUI, command line, in - IDE, browser dashboard
Focus of SA concern	Code quality, safety, security, 64- bit, other
Buildable system required	Complete build environment needed to only source code needed
Tool reputation	Small project on GitHub to Coverity
Country of origin	U.S. or U.S. allies, Russia, others
Ease of configuration and use	Simple to guru - friendly
Modification/Life Cycle	Tool end-of-lifed to multiple releases a year (w/ user bugs fixed and new features)

- No SA tool is right for all situations. A variety of considerations must be addressed.

Static Analysis Tools

We currently utilize these static analysis tools in our code assessments:

TOOL	VENDOR	UTILIZATION
Understand	Scientific Toolworks	Metrics, dependencies, code browser
Unified Code Count (UCC)	Open source	SLOC, complexity, maintainability index
PC-lint Plus	Vector	General code diagnostics
CppCheck	Open source	General code diagnostics
Parasoft C/C++test	Parasoft	Security and general code diagnostics
Lattix Architect	Lattix	Architecture and dependencies
GCC and Clang	Open source	General code diagnostics
Visual Studio Code	Microsoft	Code editor, search, browser
Sublime Text	Sublime	Code editor, search, browser

We use these tools, but SEI does not endorse any specific tool or vendor.

Assessing Code Quality and Security

Code Analysis

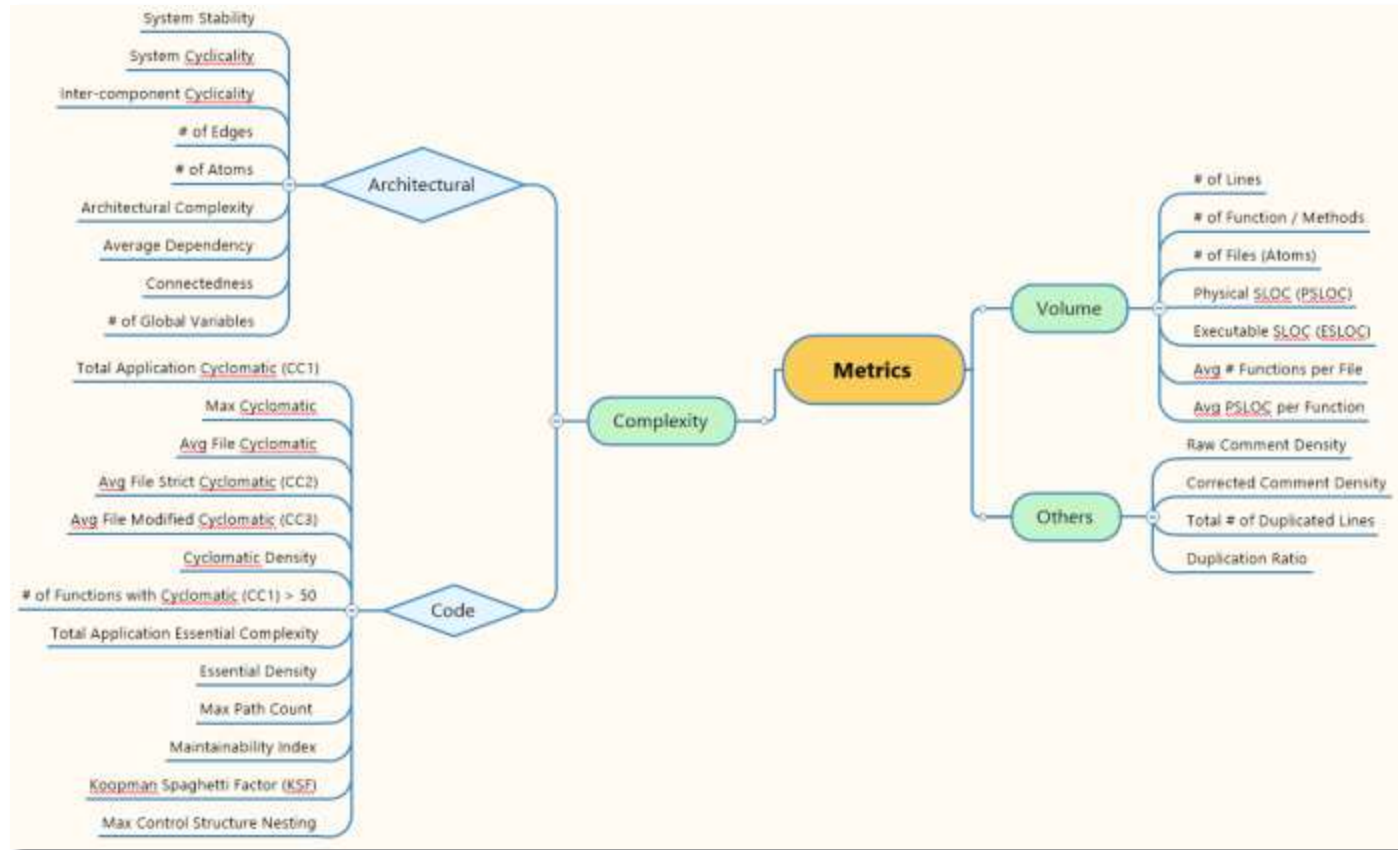
Structural Metrics

Jay Marchetti

Structural Metrics

Structural Metrics are measures or counts of certain characteristics of source code that provide insight into a variety of QA's.

Some useful metrics that SEI considers are →



Volume

Source Lines of Code (SLOC) is a rule-based count of the number of lines in a program's source code. It is the most familiar structural code volume metric.

SLOC is sometimes controversial because it can be misused...

Misuse	Use
Misusing SLOC as a <u>productivity metric</u> incentivizes code bloat and fails to capture differences in: • Language – ASM versus C/C++ • Code complexity – Naturally complex code • Developer expertise – The best developers may actually write fewer lines of (superior) code and / or be assigned more complex modules	Correlates with total system complexity and many other related and important concerns [6], e.g.: • Maintenance costs • Migration costs • Undetected error counts • New developer training time • New feature costs • Required development resources

SLOC is akin to mass. “The greater the SLOC the more inertia to change the system will have, the more people it will take to keep it fed, and the more stakeholders who will be crawling all over it.”[Grady Booch]

Complexity

There are two ways a software-intensive system exhibits complexity.

- Both render the system difficult to manage, maintain, modify, reuse, or test.

Code Complexity

The most common metric for code complexity is McCabe's Cyclomatic Complexity (CC), which is a count of the number of linearly independent paths through the code.

Architectural Complexity (AC)

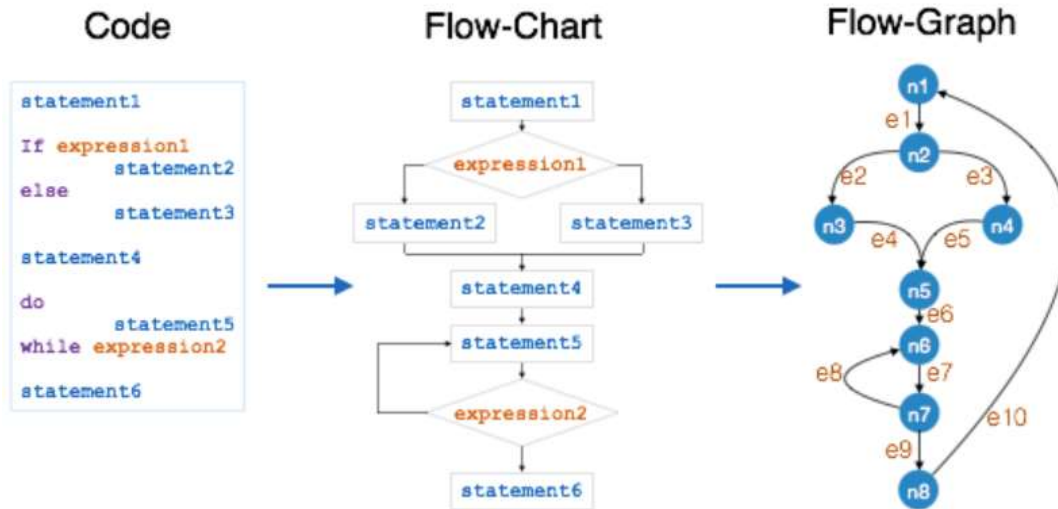
A relative measure of the number of classes/objects, processes/threads, components, and nodes, along with the quantity and type of interconnections between them [6].

Software Entropy → Without energy input (i.e., maintenance, refactoring), a system's complexity will increase over time toward chaos, with respect to both architecture and code. This accumulation of disorder over time is a form of **Technical Debt**.

Cyclomatic Complexity (CC)

CC is the number of linearly independent paths through the code and is based on the code's control flow graph (CFG).

CC may be calculated for an entire application, but more usefully it is calculated for a function module since CC equals the number of test cases required to unit test each independent code path [9]. This test coverage is also known as Basis Path testing.



$$v(G) = e - n + 2$$

where [9]

e = # flow-graph edges

n = # flow-graph nodes

Here:

$e = 10$

$n = 8$

$$CC = 10 - 8 + 2 = 4$$

Shortcut: $CC = \# \text{ CFG enclosures} + 1$

High Cyclomatic Complexity is a Red Flag

The primary importance of CC is as an indicator of the code's testability.

- As the number of decision branches rises, the CC rises too, to where “High cyclomatic numbers of more than 50 imply the code cannot be tested...” [Reliability Information Analysis Center].

Critical System Standard	Cyclomatic Limit
MISRA (Report 5 - Metrics)	≤ 15
Joint Strike Fighter (JSF++)	≤ 20
DO-178C (for FAA certification)	≤ 10
Hersteller Initiative Software (HIS)	≤ 10

Most critical coding standards limit the CC of all functions within an application.

- High CC is a **red flag** for code deserving closer review, refactoring, and additional testing.
 - High CC code is hard to test thoroughly & unthoroughly tested code is less reliable.
 - The CC metric is widely used for Quality Assurance and test planning purposes. [RIAC]

Agenda

Assessing Code Quality and Security

- 4-hour (total) course
- Broken into three 75-minute parts
- 5 to 10-minute break between parts
- Optional Q&A time at the end



Part	Duration (minutes)	Topics	Discussing
1	10 15	Introduction Software Quality	• Course Introduction and Overview • Quality Attributes • Code and Quality Standards
	50	Code Analysis	• Static and Dynamic Analysis • Static Analysis Tools • Structural Metrics
2	75	Code Risk Estimation Worksheet (CREW)	• Introduction, Overview, and Objectives • Inputs, Process Flow, and Outputs • Code Quality Aspects • Analyst Questions and Scoring Risk • Supporting Analytics (Scripts)
3	40	CREW	• Tab Walkthrough
	20	SCALe	• Overview of SCALe Security Analysis
	15	Conclusion	• Briefing CREW Results • Recommendations for Acing CREW • Potential Future CREW Improvements

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

Introduction, Overview, and Objectives

Jay Marchetti

Poor Code Quality: So, what are we up against?

```
1  #include <string.h>
2
3  int G;
4
5  size_t Function1(int a, void* vptr, char* userinput) {
6      float* ptr1, ptr2;
7      float x, a = 6.023E+23;
8      char s[12];
9      ptr1 = (float *)vptr;
10     if (*ptr1 > a)
11     {
12         x = x + 3.1415927 * a + (float)G;
13     }
14     else {
15         // G = a;
16         strcpy(s, userinput);
17     }
18     return strlen(s);
19 }
```

	Issue
●	Magic numbers
●	Uninitialized variables
●	Dead code
●	Commented-out code
●	Vulnerable library calls
●	Unvalidated inputs
●	Global variables
●	Stylistic inconsistency
●	Lack of commenting

Code Quality Analysis – Context

Within SEI, our EMCS code analysis team assesses the risk of poor code quality:

- For an existing code base – i.e. not greenfield development
- For critical systems – whether mission-critical, safety-critical, or security-critical
- For an embedded system – C or C++ typically on an RTOS or Linux platform
- Code-only – with or without requirements, design, or other documentation
- SMElessly – assuming minimal or no access to developers or system SME's
- Multi-tooled – using multiple static analysis tools, metrics, and critical-system standards
- Eyes-on-code – including the analyst's reading of a subset of the code base
- For classified or unclassified code

The goal: Considering just the source code, estimate its degree of quality risk.

Code Quality Analysis - Objectives

Create a simple framework to guide our estimation of source code quality risks, where:

- The analyst is presented with code-level questions to consistently direct the analysis
- Questions are answerable using static analysis (SA) tools and by reading the code
- The questions are intended to be SA tool agnostic
- The framework is directly applicable to critical embedded systems coded in C/C++
- Detailed code-centric questions aim to objectify the risk estimation as much as possible
- Risk is estimated statistically – i.e. no question or metric is “make or break”
- Questions of differing types can be scored and combined to derive a resulting risk

The Code Risk Estimation Worksheet (CREW)

SEI implemented code quality risk estimation within a single MS Excel worksheet, where:

- The approach is based originally upon code assessments done by SEI supporting the US Navy's Zumwalt-class guided missile destroyer
- Risks are assigned within the interval [1.0, 5.0]
- Each aspect or focus grouping within an aspect has an associated analyst tab containing the relevant detailed code-oriented questions
- There are about 500 analyst questions (total), each with a relative weighting, posed as:
 - Requested metrics values from static analysis tools or as calculated by the analyst
 - Requested quantitative estimates from the analyst
 - Qualitative analyst inputs from a drop-down selection list
- Analyst responses are automatically normalized and combined into a [1.0, 5.0] risk value for each focus
- The calculated foci risks are rolled up for each aspect and for an overall risk estimate

Customer: "Tell me about the risk level of my code."
SEI: "OK, let me ask you 500 questions about it."

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

Inputs, Process Flow, and Outputs

Jay Marchetti

Code Quality Risk Analysis - Inputs

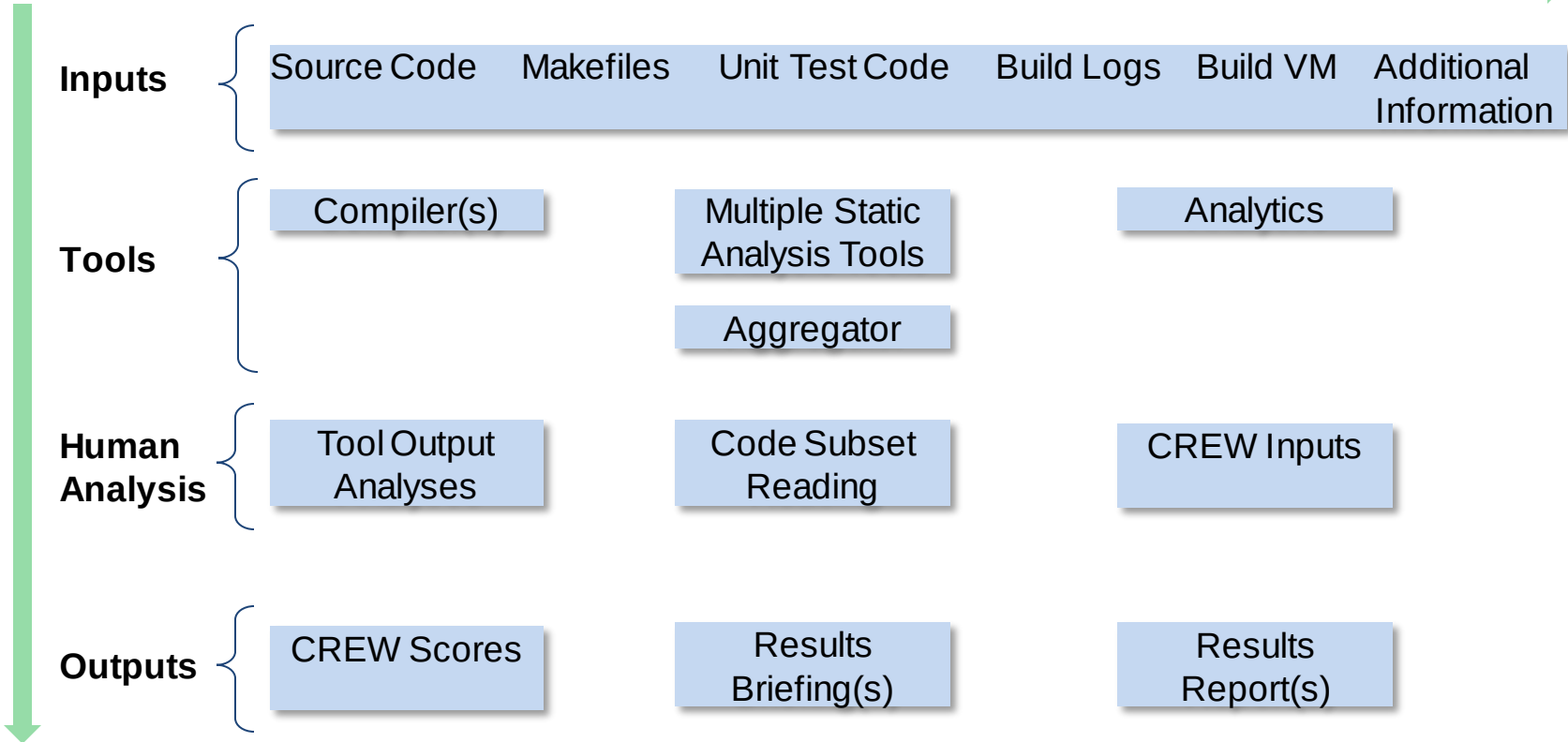
High-priority items needed to conduct the analysis

Nice to have, if possible

#	Priority	Received	Artifact	Note
1	High	Yes	All source code needed to build the system organized in the proper subfolder tree for the build.	
2	High	Yes	All build scripts, project files, solution files, or makefiles needed to build the system executable(s) from the supplied source code files.	
3	High	No	Specification: Build Instructions	
4	High	Yes	A complete build log showing the build results from / verifying the build.	
5	High	Yes	A complete list, including precise version numbers, for all compilers, linkers, target RTOS's / OS's, build host OS's, build host IDE's, preprocessing tools, loaders / debuggers, and any other utilities used to build the system executable(s).	
6	High	Yes	All unit test code, mocks, and supporting unit test framework code.	
7	High	No	The precise (i.e. manufacturer's part number) of the processor and board (if COTS) for which the executable(s) are targeted.	
8	High	No	A complete list, including precise version numbers, for all code generation tools utilized.	
9	High	No	A virtual machine (VM) .OVF file containing the entire source, build system, and all required tools that demonstrates / verifies the build.	
10	Medium	Yes	Specification: Concept of Operations (CONOPS)	
11	Medium	No	Specification: Software Development Plan (SDP)	
12	Medium	No	Specification: System Architecture Specification (SAS)	
13	Medium	Yes	Specification: Software Design Specification(s) (SDS)	
14	Medium	No	Specification: Software Test Specification (STS)	
15	Medium	No	Specification: Software Deployment Plan for executable(s) installation & upgrades	
16	Medium	No	Specification: Coding Standard (custom or adopted)	
17	Low	No	Other relevant process documents utilized for the system including: References for critical system coding standards used, a list of any static analysis tools employed, a list of any version control tool(s) and documentation of the branching strategy used, a list of any defect tracking tool(s) used.	
18	Low	No	Files identifying the current prioritized Problem Report (PR) backlog and the six (6) most recent months of PR burn-down trend data.	
19	Low	No	Any linkable library files needed (board support, RTOS, peripherals, etc.) to build the system executable(s) from the supplied source code files.	
20	Low	No	The system executable(s) built from the supplied source code.	

Code Quality Risk Analysis

Process Flow



Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

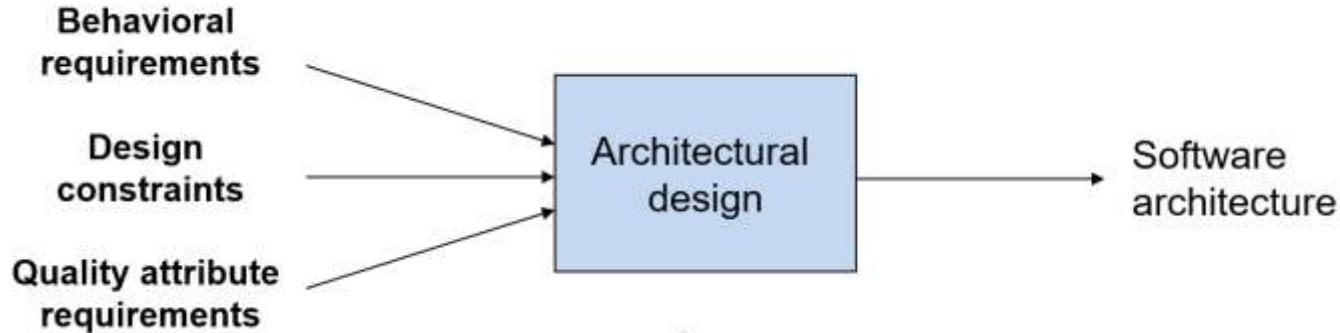
Six Assessment Aspects

Jay Marchetti

Architecture

The **software architecture** of a system is the set of structures needed to reason about the system, including software elements, relations among them, and properties of both [2].

- Every software system has an architecture, whether it is known and documented or not.
- A software architecture is not *inherently* good or bad, however...
 - Certain architectural characteristics tend to make a system more understandable, maintainable, testable, and reusable.



Standards

Standards are collections of coding rules, guidelines, and best practices that help to ensure code quality.

Adherence to standards will greatly improve the likelihood that the resulting software embodies the most commonly important QA's, i.e. that your code is:

- Maintainable throughout the system's intended life
- Consistent and Understandable
- Safe and Secure
- Testable and Reliable

Maintainability

Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [17].

Maintainable code allows others (or your future self) to easily understand, support, modify, and reuse it over its full intended lifetime. Maintainable code:

- Uses comments and a concise naming convention consistently and effectively
- Minimizes the complexity of code, classes, file & folder organization, and the dependencies among them
- Follows a written coding standard consistently
- Contains no duplicated or dead code
- Is readily human-readable and understandable

Fault Tolerance

Fault Tolerance reflects the ability of software to detect and recover from faults, whether they are external (e.g. failed h/w or improper inputs) or internal (e.g. incorrect code).

Fault tolerant code safeguards proper operation and takes appropriate action based on the nature of the failure and its essential quality attributes (QA).

For example, a safety-critical (QA = Safety) application must transition to a safe state, while a software system that must remain operational (QA = Availability) may require redundant elements. In general, fault tolerant code:

- Verifies all inputs and outputs, both at the system's perimeter and internally
- Uses consistent error handling and logging
- Continuously monitors itself and external components for proper operation

Testability

Testability is the degree to which a system or component facilitates a) the establishment of test criteria and b) the performance of tests to determine whether those criteria have been met [17].

Essentially, the testability of a program P is the probability that if P contains faults, P will fail under test [18].

Testable code is highly modular, self-contained, and simple. More specifically, it exhibits:

- High cohesion – has a singular and well-defined purpose
- Low coupling – has few external dependencies
- Low complexity – has few decision paths

Security

Security of code consists of techniques to protect a system from known or novel vulnerabilities so that it continues to function as intended and remains safe from attacks.

Secure code is exemplified by adherence to secure coding standards such as the SEI CERT Secure Coding standards.

More generally, secure code embodies accepted best practices regarding:

- Defensive coding
- Validation and encryption
- Libraries and API's
- Traceability of executable and data files

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

Analyst Questions

Jay Marchetti

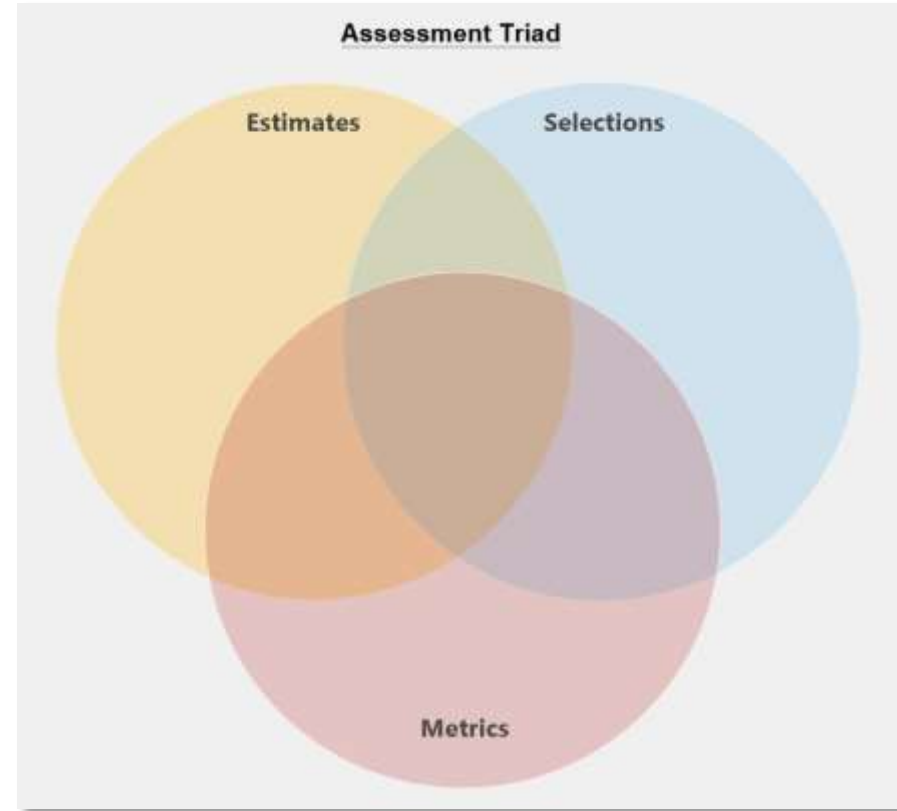
CREW Analyst Questions – Three Types

CREW poses 3 different types of code-centric queries to the analyst.

The analyst may utilize SA tools, analytics (scripts), and / or source code reading in order to answer them.

Type	Description
Selection	Subjective drop-down menu choice
Estimate	Informed valuation
Metric	Tool output or calculated measurement

The answers are combined to form an overall code quality risk estimate score.



CREW Analyst Questions – Examples

Analyst tab inputs are: metrics, estimates, or multiple-choice selections.

UTILIZATION	4.6	17	1	Estimated total number of commented-out lines of code	326	0
		18	1	Are commented-out lines interspersed freely among uncommented-out lines of code?	Sometimes	
		19	1	Estimated total number of unreachable statements within the code	No	
		20	1	Estimated total number of unused functions or methods within the code	Seldom	
		21	1	Estimated total number of unused global variables within the code	Sometimes	
		22	1	Estimated total number of unused local variables within the code	Often	
COMPLICATION	3.9	23	1	Estimated total number of no-effect or redundant statements within the code	Yes	
		24	1	Maximum control structure nesting level within the code	12	0
		25	0	Total number of functions or methods within the code base	2842	0
		26	1	Percentage of functions or methods within the code with Hersteller (HIS) VOCF (LanguageScope) equal to or exceeding 4.0	0.5%	0
		27	1	Number of goto occurrences within the code, i.e. Hersteller GOTO metric	13	0
		28	1	Number of functions within the code with Hersteller AP CG_CYCLE (Recursion) of 1 or more	3	0
		29	1	Percentage of functions or methods within the code with Hersteller CALLING (FanIn) exceeding 5	0.7%	0
		30	1	Percentage of functions or methods within the code with Hersteller CALLS (FanOut) exceeding 7	1.3%	0
		31	1	Percentage of functions or methods within the code with Hersteller PARAM (NumParams) exceeding 5	14.1%	0
		32	1	Percentage of functions or methods within the code with Hersteller LEVEL (Nesting) exceeding 4	0.2%	0
DUPLICATION	3.7	33	1	What is the level of duplicated code lines, i.e. how much copy / pasted code is there?	Some	0
		34	1	Estimated percentage of source files (i.e. file count) that contain any duplicated code	2.3%	0
		35	0	Lines of code (PSLOC)	231741	0
		36	1	Total number of lines of duplicated code detected	5300	0
		37	1	Duplication ratio (as a percentage) = total number of detected duplicated lines to PSLOC	0.3%	0

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

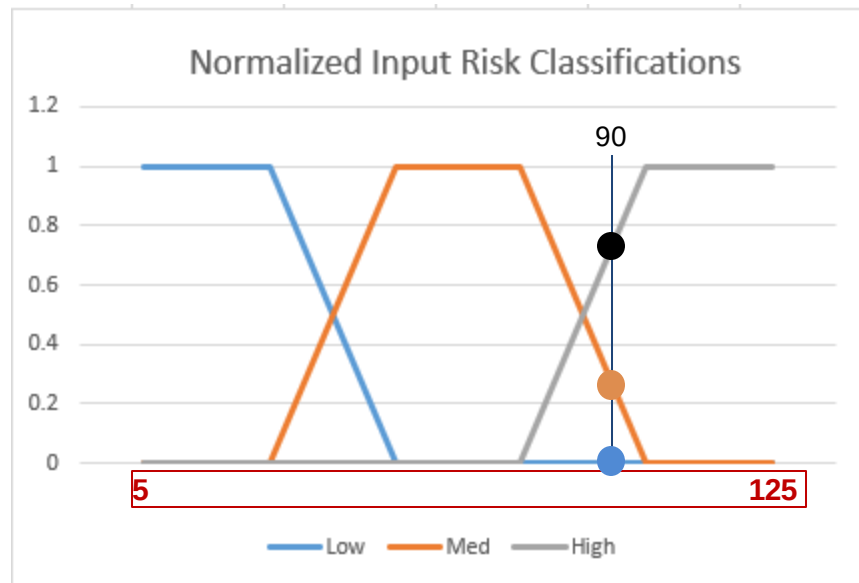
Scoring Risk

Jay Marchetti

Scoring Risk

How can we combine the 3 disparate types of analyst tab inputs (Selections, Estimates, and Metrics) into a unified risk assessment?

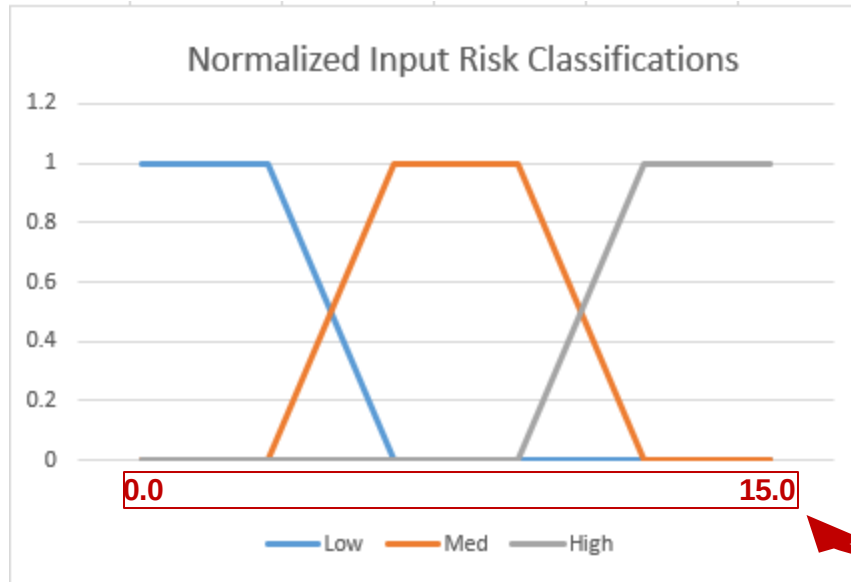
- Using concepts motivated by Fuzzy Logic, we can normalize analyst answers and then combine them using 3 risk membership functions: Low, Medium, and High.
- The abscissa range for each question is linearly transformed to accommodate the answer range.
- Then the answer is expressed as an amount of each risk component: Low, Medium, and High.
- Last, the 3 components are summed across all questions for a given Focus to calculate an aggregate scalar risk value for that Focus.



This normalization and aggregation is computed automatically within CREW.

Scoring Risk – Normalization

Analyst answers are normalized by a simple linear transformation of the membership function abscissa (x-axis) range that is specific to each question.



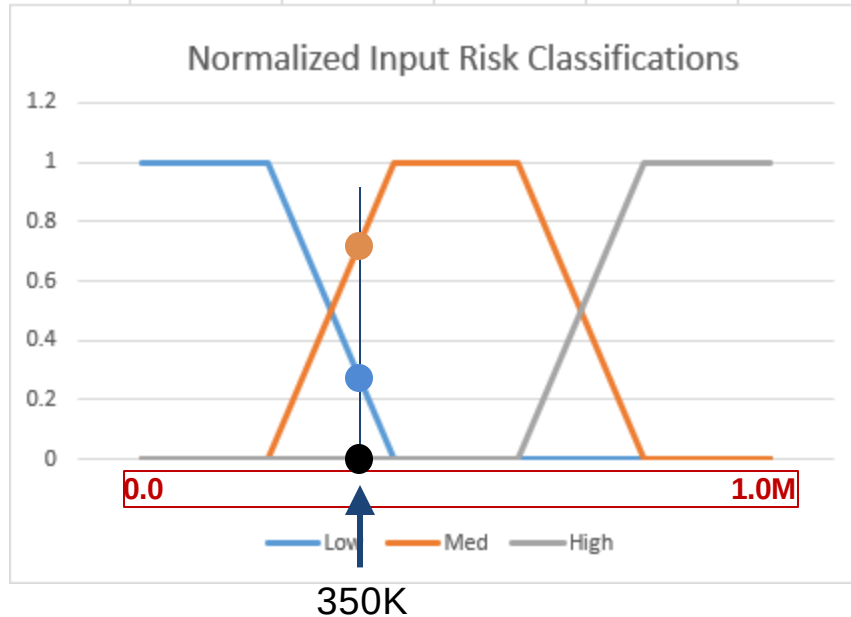
For each question, the values of the X-axis are gained and offset as $x' = mx + b$ in order to compute the risk classification memberships.

Example #1: For Cyclomatic complexity (CC) we may use $m = 15$ and $b = 0$ for a range of 0.0 to 15.0.

★ It is these 2 parameters, gain (m) and offset (b), that set the acceptable “degree of risk” for each numerical analyst input. E.g. $CC < 15$.

Scoring Risk – Normalization₂

Each analyst's answer is applied to the transformed classification membership functions to compute the 3 components (Low, Med, High) using the input's intercepts.



Example #2:

For SLOC let's assume that gain (m) = 1.0E6 (1M) and offset (b) = 0.0.

Suppose a static analysis tool tells the analyst that the code metric SLOC = 350K.

Then the 3 risk component membership values will be:

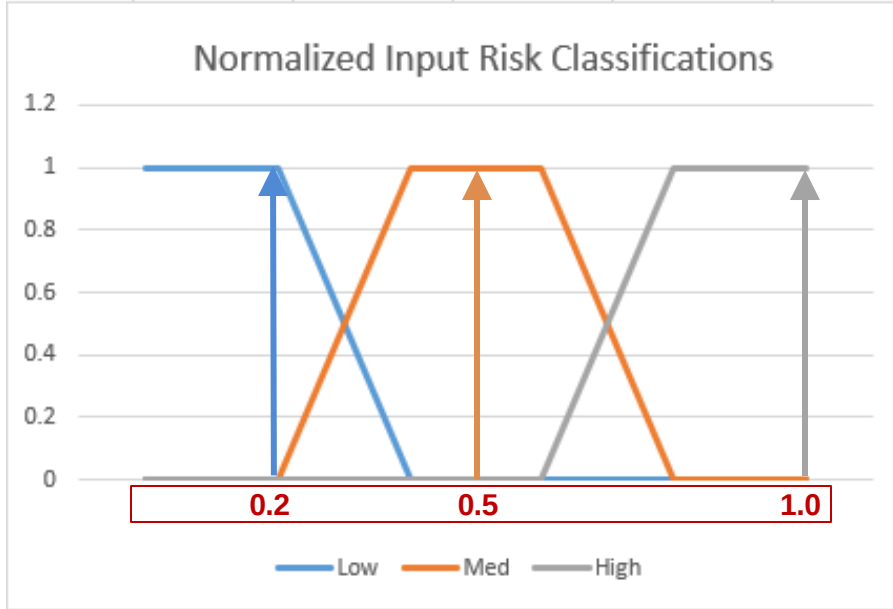
- Low = 0.25
- Med = 0.75
- High = 0.00

Note that the risk components always sum to 1.00

Scoring Risk – Aggregation to a Crisp Output

Finally, we calculate the average of the components (Low_{AVE} , Med_{AVE} , $High_{AVE}$) for all the analyst answers within each Focus.

- We then compute their dot-product with 3 singletons positioned at the input member set centers to yield a crisp composite risk for each Focus.



Note that we compress the crisp result (and multiply by 5.0) so that we obtain final crisp risk values within $[1.0, 5.0]$.

This is done by using a left-most singleton value of 0.2 rather than 0.0, i.e.

$$\text{Risk} = (Low_{AVE} * 0.2 + Med_{AVE} * 0.5 + High_{AVE} * 1.0) * 5.0;$$

Scoring – Focus Risk & Weights and Aspect Roll-up

Example: Testability Tab

Focus	W	#	I	Measurement / Question	A	Finding
UNIT	2.6	0.350	1	Does the supplied code base include the unit test code?		Yes
			2	Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?		Somewhat
			3	Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?		Some
			4	Do unit tests utilize assertions liberally to indicate test success or failure?		Some
			5	Does the coverage of the unit tests appear to be comprehensive?		Many
			6	Does the code base utilize global variables?		Minimal
			7	Does the test code allow easy documentation of application or library code?		No
			8	Do tests utilize both positive and negative test cases, i.e. fault injection?		Somewhat
			9	Are the unit tests generally easy to understand and verifiable?		No
REGRESSION	2.0	0.350	10	Is there evidence in either code or comments that the code is regression tested?		Some
			11	Is there evidence of test driver usage or other automated regression testing?		
			12	Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?		
COMPLEXITY	3.9	0.350	13	Maximum cyclomatic complexity (CC1) for any function within the code base	•	100
			14	Total number of functions or methods within the code base	•	5000
			15	Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10		80.0%
			16	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20		8.0%
			17	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30		6.0%
			18	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40		4.0%
			19	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50		1.8%
			20	Number of functions within the code with cyclomatic complexity (CC1) > 50		12
			21	Average file cyclomatic complexity (CC1)	•	3.33
			22	Average file strict cyclomatic complexity (CC2)		4.46
			23	Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10		25.0%
			24	Average file modified cyclomatic complexity (CC3)		3.00
			25	Percentage of functions or methods within the code with Hersteller PATH exceeding 80		3.7%
RISK	3.44					

Drives

- The analyst's answers are scored to determine the Focus risk.
- Each Focus has a relative weight. These sum to 1.00 for each tab.
- The Aspect risk is then calculated as the weighted average of the Focus risks.

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

Supporting Analytic Scripts

Alan Cohn

Current Supporting Analytic Scripts

We currently utilize these supporting scripts to aid with our code assessments:

SCRIPT NAME	DESCRIPTION
Maintainability Comments	Aids with the maintainability comment CREW tab
Koopman Spaghetti Factor (KSF)	Determine file complexity based on SCC, globals, and SLOC
Conditional Compiles	List and provide metrics for conditional compiles
Keyword Search	Search for various categories of keywords in the code
Assertion Density	Identify and calculate metrics related to assertions in the code
Includes Together	Identify files with includes not grouped at the top of the file
Whitespace Analysis	Provide metrics on leading whitespace and tabs
Duplicate Code	Understand plug-in + python to identify duplicate lines and files

These are the completed SEI scripts. New scripts and enhancements are planned.

Support Analytic Script Example – Maintainability Comments

```
Please pick one of the following options:
1. Get Module Header and Comment Information -> Analyze all files and get the distribution/metrics of file headers and comment make up
2. Perform Keyword Search -> Search the code for various sets of keywords
3. Get Total Number of Variable Declarations -> Get the total number of global and local (file and function) declarations
4. Get Number of Comments before Classes or Functions -> Calculate metrics for classes, functions, and methods and preceding prologs
9. Exit

*****
User Entry: █
```

```
Number of standard '//' comments: 21463
Number of Single lined C-Style '/* ... */' comments: 9
Number of block comments spanning more than one line: 3372
Number of lines with comments in line with code: 1873
Number of comments without a preceeding blank line: 9368
Number of comments not aligned with the next line: 7372

**** HEADER DISTRIBUTION ****
HEADER LENGTH : NUMBER OF FILES
33 : 1248
31 : 36
29 : 1
34 : 3
88 : 24
21 : 2
23 : 4
25 : 2
15 : 3
17 : 1
24 : 1
22 : 1

Total Number of Files processed: 1320
Total Number of Module Header Lines: 42537
Percent of files with module header block: 98.18
Average Lines Per Block: 32.82
```

```
Number of Functions Processed: 81174
Number of Functions with Preceeding Comment: 4482
Percentage of functions that have a preceeding comment: 5.52

Number of Classes Processed: 3049
Number of Classes with Preceeding Comment: 0
Percentage of classes that have a preceeding comment: 0.0

TOTAL PERCENTAGE OF CLASSES AND FUNCTIONS WITH A PRECEEDING COMMENT: 5.32
```

Maintainability Comments/Quantity	6	1	Estimated percent of source files containing a module header block	98%
Maintainability Comments/Quantity	7	1	Are existing module header comment blocks merely copyright info?	Yes
Maintainability Comments/Quantity	8	0	Approximate number of comment lines per existing module header block	32.18
Maintainability Comments/Quantity	10	1	Estimated percent of class, function, and method declarations that are commented	6%
Maintainability Comments/Quality	16	1	Are code modules separated into standardized sections via comments?	None
Maintainability Comments/Quality	17	1	Are C-style '/* ... */' comments used?	Some
Maintainability Comments/Quality	18	1	Are // comments used?	Some
Maintainability Comments/Quality	19	1	Are end-of-line comments used?	Some
Maintainability Comments/Quality	20	1	Are block comments spanning multiple lines used?	Some

Support Analytic Script Example – Koopman Spaghetti Factor

$$\text{KSF} = \text{COMPLEXITY} + (\text{GLOBALS} * 5) + (\text{SLOC}/20)$$

THE AVERAGE KSF FOR ALL FILES PROCESSED (657) IS: 29

File	Complexity	Globals	Lines	KSF
../communications/iop_bridge/iop_messages/src/ugv/ReportPowerPlantStatus.cpp	15	0	3443	187
../communications/iop_bridge/iop_messages/src/ReportDiagnostics.cpp	55	0	2529	181
../iop_messages/src/manipulator/ReportManipulatorSpecifications.cpp	11	0	3363	179
../communications/iop_bridge/iop_messages/src/custom_rtk/SetCommsLostPolicy.cpp	10	0	3229	171
../communications/iop_bridge/iop_messages/src/iop/SetCommsLostPolicy.cpp	10	0	3124	166
../iop_messages/src/environmentSensing/ReportVisualSensorCapabilities.cpp	19	0	2800	163
../communications/iop_bridge/iop_messages/src/ugv/ReportPowerPlantCapabilities.cpp	11	0	2798	150
../communications/iop_bridge/iop_messages/src/iop/ReportCommunicatorCapability.cpp	11	0	2767	149
../communications/iop_bridge/iop_messages/src/ugv/ReportPlatformSpecifications.cpp	19	0	2600	141
../communications/iop_bridge/iop_messages/src/iop/ReportCommsLostConfiguration.cpp	10	0	2695	144
../communications/iop_bridge/iop_messages/src/iop/ReportPresetPoseSpecifications.cpp	8	0	2403	128

FILES WITH KSF LESS THAN 10: 63
 FILES WITH KSF LESS THAN 20: 190
 FILES WITH KSF LESS THAN 30: 184
 FILES WITH KSF LESS THAN 40: 79
 FILES WITH KSF LESS THAN 50: 48
 FILES WITH KSF GREATER THAN 50: 93

UNUSED GLOBAL DATA:

TOTAL NUMBER OF UNUSED VARIABLES: 0

USED GLOBAL VARIABLES:

TOTAL NUMBER OF USED GLOBAL VARIABLES: 0

Maintainability Reusability/Complication	33	1	Average Koopman Spaghetti Factor (KSF) of all implementation files	29.0
Maintainability Reusability/Complication	34	1	Percentage of implementation files with KSF exceeding 15	66.0%
Maintainability Reusability/Utilization	21	1	Estimated total number of unused global variables within the code	0
Testability/Unit	6	1	Does the code base utilize global variables?	No

Agenda

Assessing Code Quality and Security

- 4-hour (total) course
- Broken into three 75-minute parts
- 5 to 10-minute break between parts
- Optional Q&A time at the end



Part	Duration (minutes)	Topics	Discussing
1	10 15	Introduction Software Quality	• Course Introduction and Overview • Quality Attributes • Code and Quality Standards
	50	Code Analysis	• Static and Dynamic Analysis • Static Analysis Tools • Structural Metrics
2	75	Code Risk Estimation Worksheet (CREW)	• Introduction, Overview, and Objectives • Inputs, Process Flow, and Outputs • Code Quality Aspects • Analyst Questions and Scoring Risk • Supporting Analytics (Scripts)
3	40 20 15	CREW SCALe Conclusion	• Tab Walkthrough • Overview of SCALe Security Analysis • Briefing CREW Results • Recommendations for Acing CREW • Potential Future CREW Improvements

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Jay Marchetti

Navigating the CREW Analyst Tabs

Example: Architecture Tab

The Analyst edits only these 3 things:

- Finding
- **D** or “Done” marker
- Optional text Note

I or “Included in scoring” indicator

A or “Auto” marks (•)
System tab / pre-calc'd or
(C) conditional questions

The Modularity

Focus has:

- 7 included questions to answer (#2 – #8)
- 0.333 Weight for this Aspect
- A resulting **Risk** score of 2.1

The resulting weighted
Architecture **Aspect**
risk score of 2.72

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
MODULARITY	2.1	0.333	1	0	Atom count (# elements in the system)	• 800		1	
			2	1	Is there a clear separation of functional components within the system?		Yes	1	
			3	1	Is there a high degree of coupling among components?		Minimal	0	
			4	1	Within components is there a high degree of coupling among modules?		Somewhat	0	
			5	1	System stability (% of elements (on avg) that would not be affected by a change to an element)		90.0%	0	
			6	1	Minimum component stability		65.0%	0	
			7	1	Connectedness (% ordered pairs of systems (A,B) such that A directly or indirectly depends on B)		2.0%	0	
			8	1	To what degree are global variables utilized within the system?		Many	0	
DEPENDENCY	2.4	0.333	9	1	From the dependency graph, how many dependencies are bidirectional?		Most	1	
			10	1	How many dependencies are there on COTS or FOSS components?		None	1	
			11	1	Total number of internal dependencies (total # of edges between all atoms)		1000	1	
			12	1	Average dependency (total edges / total # atoms)	• 1.25		0	
			13	1	Architectural complexity (# atoms multiplied by # edges / 10.0E+06)	• 0.08		0	
			14	1	System cyclicity (% elements in the system that are part of a cycle)		9.5%	0	
			15	1	Intercomponent cyclicity (% elements in a cycle with systems in other components of the arch)		5.1%	0	
			16	1	Does the system use a common architectural pattern (layering, client-server, MVC, etc.)?		No	1	
PATTERNS	3.7	0.333	17	0	Are any COTS or FOSS components cleanly separated from custom application code?	C	Somewhat	1	
			18	0	Does the system use the most commonly found architectural pattern: layering?	C	Yes	1	
			19	0	If layered, is the system strictly layered (calls are only to the adjacent lower layer)?	C	No	1	
			20	0	Is the architectural pattern used evident in file and subfolder naming and organization?	C	No	1	
			21	1	Do architectural boundaries exist that hide implementation details within components?		Mostly	1	
RISK	2.72								

CREW Tab Walkthrough

Code Risk Estimation Worksheet
v3.0

Project: SystemX
Last Revision: 1.1.2
Language(s): C
Run-time OS: Windows 6.0
Run-time CPU: Unknown
Analyst(s): Albrecht, Brown, Rodin, Cook, Bookout
Review Date: 05/14/23

USAGE KEY

- Aspect: The six (6) aspects of code quality being assessed for risk.
- Focus: Sub-aspects that are rated via the nine (9) scoring tabs.
- Best Practice: Focus best practice / positive code quality characteristic.
- Change: Optional analyst's noted change in a rating from the assessment of a prior code drop (input [-] / delta).
- Analyst: Optional assigned analyst for an aspect (input [last]).
- Notes: Optional additional analyst's notes (input [text]).
- Rating: Rolled up - risks from the nine (9) details tabs that the integrative statements are NOT true.

1.0 (Low Risk) 3.0 (High Risk)

CODE RISK ESTIMATION WORKSHEET FOR: SystemX

ASPECT	RISK	ASPECT / Focus	BEST PRACTICE	CHANGE	ANALYST	NOTES
2.4	ARCHITECTURE	Modularity	High level of code modularity and module cohesiveness		Albrecht	
2.4	ARCHITECTURE	Dependency	Low level of dependencies, especially cyclic (DAG, dependency graphs, etc)			
2.4	ARCHITECTURE	Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)		Bookout	
2.4	STANDARDS	Style	High level of adherence to a chosen internal or published stylistic coding standard			
2.4	STANDARDS	Complexity	High level of adherence to critical system coding guidelines (MISRA, CERT, etc)			
2.4	STANDARDS	Language	High level of adherence to widely accepted language guidelines			
2.8	MAINTAINABILITY	Comments	High level of commenting excluding boilerplates		Bookout	
2.8	MAINTAINABILITY	Clarity	High quality of commenting that clarifies the code's intent			

Risk Results Inputs System Architecture Standards MaintainabilityComments MaintainabilityClarity MaintainabilityPersonality FaultTolerance Testability SecurityGeneral SecurityCERT

So now, let's walk through each tab in CREW...

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Inputs

Jay Marchetti

CREW Tab - Inputs

#	Priority	Received	Artifact	Notes
1	High	Yes	All source code needed to build the system organized in the proper subfolder tree for the build.	
2	High	Yes	All build scripts, project files, solution files, or makefiles needed to build the system executable(s) from the supplied source code files.	
3	High	No	Specification: Build Instructions	
4	High	Yes	A complete build log showing the build results from / verifying the build.	
5	High	Yes	A complete list, including precise version numbers, for all compilers, linkers, target RTOS's / OS's, build host OS's, build host IDE's, preprocessing tools, loaders / debuggers, and any other utilities used to build the system executable(s).	
6	High	Yes	All unit test code, mocks, and supporting unit test framework code.	
7	High	No	The precise (i.e. manufacturer's part number) of the processor and board (if COTS) for which the executable(s) are targeted.	
8	High	No	A complete list, including precise version numbers, for all code generation tools utilized.	
9	High	No	A virtual machine (VM) .OVF file containing the entire source, build system, and all required tools that demonstrates / verifies the build.	
10	Medium	Yes	Specification: Concept of Operations (CONOPS)	
11	Medium	No	Specification: Software Development Plan (SDP)	
12	Medium	No	Specification: System Architecture Specification (SAS)	
13	Medium	Yes	Specification: Software Design Specification(s) (SDS)	
14	Medium	No	Specification: Software Test Specification (STS)	
15	Medium	No	Specification: Software Deployment Plan for executable(s) installation & upgrades	
16	Medium	No	Specification: Coding Standard (custom or adopted)	
17	Low	No	Other relevant process documents utilized for the system including: References for critical system coding standards used, a list of any static analysis tools employed, a list of any version control tool(s) and documentation of the branching strategy used, a list of any defect tracking tool(s) used.	
18	Low	No	Files identifying the current prioritized Problem Report (PR) backlog and the six (6) most recent months of PR burn-down trend data.	
19	Low	No	Any linkable library files needed (board support, RTOS, peripherals, etc.) to build the system executable(s) from the supplied source code files.	
20	Low	No	The system executable(s) built from the supplied source code.	

Inputs Tab

- Tracks and provides a record of which CREW inputs have been / were received for the code analysis.
- Data entry: Analyst types “Yes” or “No” in the column labeled Received.
- Preferably, all inputs marked as “High” in the Priority column are received before the analysis commences, though quite often this is not the case.

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

**CREW Tab Walkthrough
System**

Jay Marchetti

CREW Tab - System

Date:	08/24/22				
Analyst:	Marchetti				
SA Tool Used:	Understand 6.2 (Build 1100)				
System	#	Measurement / Question	Finding	D	Note
BASIS	1	System Name	SystemX	1	
	2	Build environment OS and version	Ubuntu 22.04	1	
	3	Compiler and version	gcc 9.1.0	1	
	4	Source code language	C/C++	0	
	5	Source code / system revision if known	1.52	0	
	6	Run-time OS and version if known	VxWorks 6.9	0	
	7	Run-time CPU if known	Unknown	0	
METRICS	8	Total number of source files	800	0	
	9	Number of definition (header) files	300	0	
	10	Number of implementation (code) files	500	0	
	11	Total number of (all) lines in the code base	350000	0	
	12	Source lines of code ignoring blank and comment lines (PSLOC)	150000	0	
	13	Executable source lines of code (ESLOC)	100000	0	
	14	Number of lines in the code base containing a comment (CLOC)	120000	0	
	15	Ratio of number of comment lines to number of code lines	75%	0	
	16	Total number of functions or methods within the code base	5000	0	
	17	Average file cyclomatic complexity (CC1)	3.333	0	
	18	Maximum cyclomatic complexity (CC1) for any function within the code base	100	0	
	19	Maximum control structure nesting level within the code base	8	0	
	20	Cyclomatic complexity (total CC1) of the entire code base v(G)	20000	0	
	21	Essential complexity of the entire code base Ev(G)	5000	0	
	22	Estimated total number of commented-out lines of code	200	0	

TIP: The "Finding" column is where the analyst answers the questions.

TIP: The "D" or "Done" column is used to mark which questions are completed.

System Tab

- Always the first analyst tab to be completed.
- Captures base information and metrics utilized on the other analyst tabs ("A" column).
- Does not factor directly into the code risk scores.
- Ensures better consistency across the analysis.

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Architecture

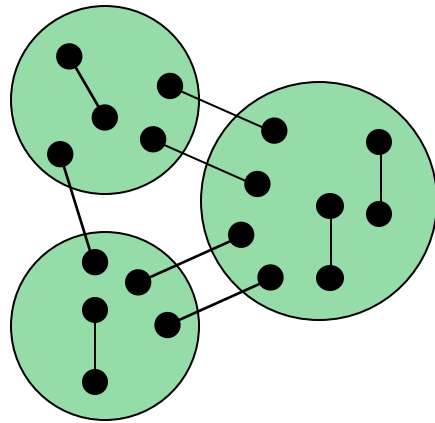
Jay Marchetti

What to Look For in Architecture

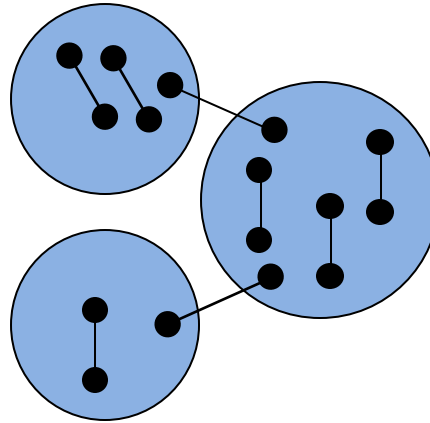
- **Modularity** - High level of code modularity and module cohesiveness
 - Is the code clearly separated into components that have unambiguous purpose?
 - Are components relatively self-contained?
- **Dependency** - Low level of dependencies, especially bi-directional and cyclic (DSM, dependency graphs, etc.)
 - Do components minimize couplings among them?
 - Do components avoid complex couplings?
 - Is the volume of modules and the inter-dependencies among them relatively low?
- **Patterns** - High level of adherence to a known architectural pattern (layered, client-server, etc.)
 - Does the system utilize a familiar software architectural pattern such as layered, message broker, service bus, etc.?

Architecture – Coupling versus Cohesion

Static Analysis can reveal the coupling and cohesion levels within a system.



System A



System B

Q: Which system is easier to maintain, reuse, and test?

A: System B 

Example at left [Adapted from 7]:
Circles are modules, lines are calls, and line ends are functions.

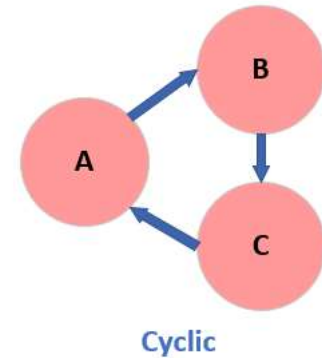
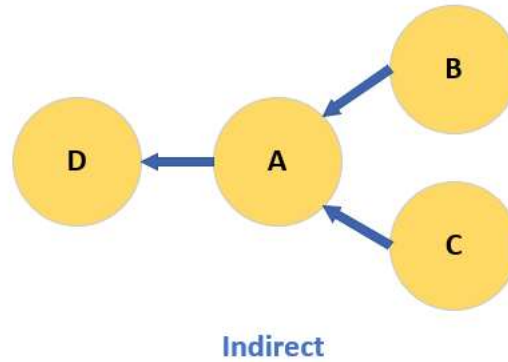
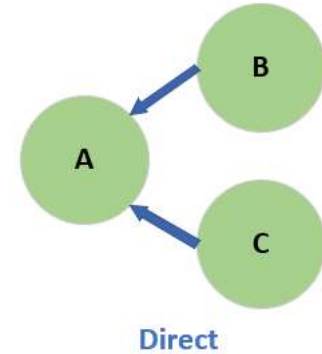
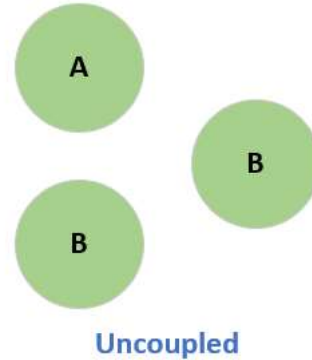
- Cohesion: calls inside a module of unified concept
- Coupling: calls between modules

	System A	System B
Cohesion	Low	High
Coupling	High	Low

Architecture – Dependency Types

Modules within a system may be dependent in several ways:

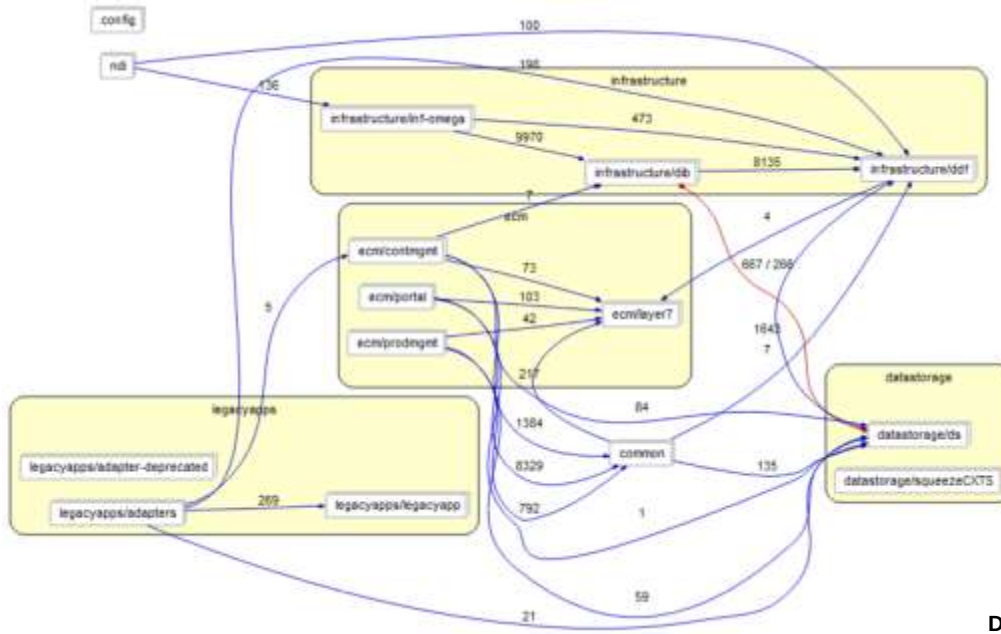
- Uncoupled
- Directly coupled
- Indirectly coupled
- Cyclic



Discerning Architecture from Code Analysis

Given a software system, where all we have access to is the source code, what can we discern purely from SA tools?

- We can look at the number and types of interdependencies among modules or classes using **Dependency Graphs** or other visualizations.



For small to medium systems, dependency graphs provide:

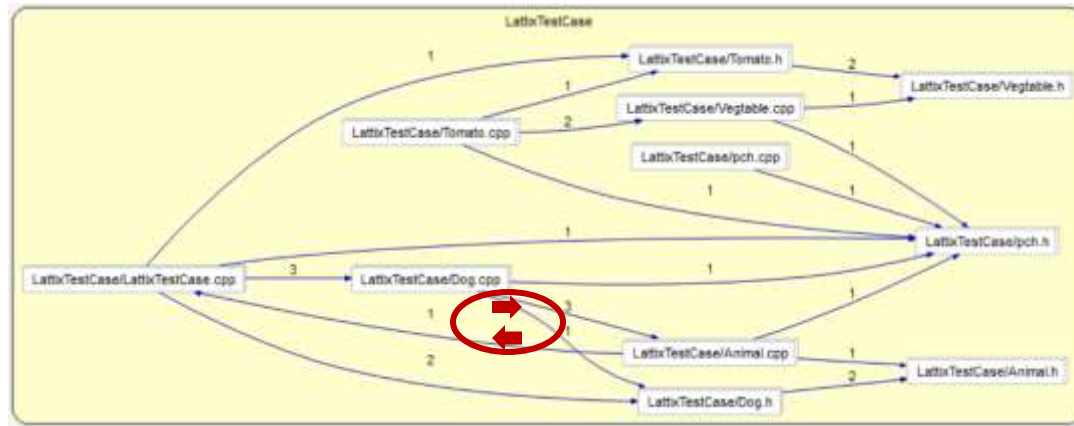
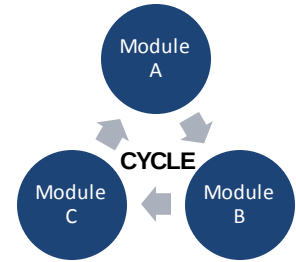
- Views of the main subsystems and their components
- Indications of the number of couplings between components and whether any are bidirectional (red)

Dependency Graph – Created Using SciTools Understand

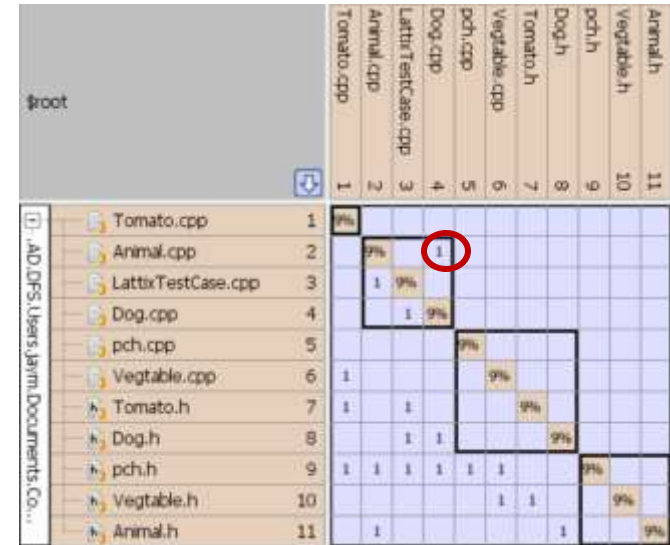
Architecture – DSM and Cyclic Dependencies

A good dependency visualization for large or complex systems is the Dependency Structure Matrix (DSM).

DSM's show cyclic dependencies (aka "cycles") as entries above the diagonal.



Dependency Graph – Created Using SciTools Understand



DSM – Created Using Lattix Architect

Two dependency views of the same system containing a cyclic dependency.

CREW Tab: Architecture (Modularity, Dependency, Patterns)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
MODULARITY	2.1	0.333	1	0	Atom count (# elements in the system)	•	800	1	
			2	1	Is there a clear separation of functional components within the system?		Yes	1	
			3	1	Is there a high degree of coupling among components?		Minimal	0	
			4	1	Within components is there a high degree of coupling among modules?		Somewhat	0	
			5	1	System stability (% of elements (on avg) that would not be affected by a change to an element)		90.0%	0	
			6	1	Minimum component stability		65.0%	0	
			7	1	Connectedness (% ordered pairs of systems (A,B) such that A directly or indirectly depends on B)		2.0%	0	
			8	1	To what degree are global variables utilized within the system?		Many	0	
DEPENDENCY	2.4	0.333	9	1	From the dependency graph, how many dependencies are bidirectional?		Most	1	
			10	1	How many dependencies are there on COTS or FOSS components?		None	1	
			11	1	Total number of internal dependencies (total # of edges between all atoms)		1000	1	
			12	1	Average dependency (total edges / total # atoms)	•	1.25	0	
			13	1	Architectural complexity (# atoms multiplied by # edges / 10.0E+06)	•	0.08	0	
			14	1	System cyclicity (% elements in the system that are part of a cycle)		9.5%	0	
			15	1	Intercomponent cyclicity (% elements in a cycle with systems in other components of the arch)		5.1%	0	
PATTERNS	3.7	0.333	16	1	Does the system use a common architectural pattern (layering, client-server, MVC, etc.)?		No	1	
			17	0	Are any COTS or FOSS components cleanly separated from custom application code?	C	Somewhat	1	
			18	0	Does the system use the most commonly found architectural pattern: layering?	C	Yes	1	
			19	0	If layered, is the system strictly layered (calls are only to the adjacent lower layer)?	C	No	1	
			20	0	Is the architectural pattern used evident in file and subfolder naming and organization?	C	No	1	
			21	1	Do architectural boundaries exist that hide implementation details within components?		Mostly	1	
RISK	2.72								

TIP: The SA tool Lattix Architect is useful for answering some of these questions.

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Standards

Alan Cohn

What to Look For in Standards

- **Stylistic** - High level of adherence to a chosen internal or published stylistic coding standard
 - Is the code written clearly?
 - Is the code formatted consistently?
- **Criticality** - High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)
 - Is the code predictable?
 - Is the possibility of crashes minimized?
- **Language** - High level of adherence to widely accepted language guidelines
 - Are standards and accepted best practices for the language followed consistently?

Analyzing Stylistic Standards

- While SA tools can be useful for identifying some violations, manual inspection of files is often required.
- CREW is designed to be able to complete an analysis with just the source code and no supporting documents. As such, the stylistic questions do not prescribe a specific coding style, but rather ensure that one is used consistently.
- If completing an internal analysis, it may be useful to check for compliance to the project's prescribed coding standard.

CREW Tab: Standards (Stylistic)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
STYLISTIC	2.5	0.100	1	1	Is each source file limited to one class or collection of routines with one and only one purpose?		Yes	0	
			2	1	Do source files headers explain their function?		No	0	
			3	1	Are includes/imports included together?		Yes	0	
			4	1	Are methods declared and defined in a logical order?		Most	0	
			5	0	Are C++ overloaded methods grouped together?	C	No	0	
			6	1	Are only spaces used for whitespace?		No	0	
			7	1	Are indentations and brace usage consistent?		Most	0	
			8	1	Does there appear to be a column limit in place for source files?		Some	0	
			9	1	Are 'paragraphs' used to separate the code into logically related sections?		Yes	0	
			10	1	Are class / module names meaningful?		Yes	0	
			11	1	Are class / module names formatted consistently?		Yes	0	
			12	1	Are method / function names meaningful?		Yes	0	
			13	1	Are method / function names formatted consistently?		Yes	0	
			14	1	Are variable names meaningful?		Yes	0	
			15	1	Are variable names formatted consistently?		Yes	0	
			16	1	Are constant names meaningful?		Yes	0	
			17	1	Are constant names formatted consistently?		Yes	0	
			18	1	Are test methods / functions consistently marked as such?		Yes	0	
			19	1	Do files include a file prologue that explains the purpose of the file?		No	0	
			20	1	Do methods / functions include a prologue comment block?		Few	0	
			21	1	Are comments formatted consistently?		Yes	0	
			22	1	Are units for variables included?		No	0	
			23	1	Are comments used to mark the ends of code blocks?		Few	0	
			24	1	Are error values formatted consistently?		No	0	

Analyzing Criticality Standards

Configuration: PowerOffen Files (5764) All Files (No Filter) Inspect Results By Checks Ignores List

48831 Violations 0 Ignored 5764 Files 10 Checks

Completed 6/30/2022 1:19 PM

Results

Entity Line Column

Number of Results: 48831

- All Checks
 - Language Specific
 - C and C++
 - Unnecessary Constructs
 - Discarded return values - CPP_U001
 - No direct or indirect recursion allowed - AC_01
 - Published Standards
 - Effective C++ (3rd Edition) Scott Meyers
 - 26 Postpone variable definitions as long as possible - EFFECTIVECPP_26
 - MISRA-C 2004
 - 17.5 no more than 2 levels of indirection - MISRA04_17.5
 - 19.7 A function should be used in preference to a function-like macro - MISRA04_19.7
 - 20.4 Dynamic heap memory allocation shall not be used - MISRA04_20.4
 - 8.7 Objects shall be local if only accessed from one function - MISRA04_8.7
 - SciTools' Recommended Checks
 - Functions Too Long - RECOMMENDED_04
 - Goto Statements - RECOMMENDED_06
 - Upper limit shall not be modified within the bounds of the loop - RECOMMENDED_17

- Many standards for safety-critical systems exist. CREW checks for compliance to a small number of basic criteria common amongst all standards and heavily influenced by NASA JPL's Power of Ten.
- Most questions are easily verified by any fully featured SA tool (Understand, SonarQube, etc.)

CREW Tab: Standards² (Criticality)

Focus	Risk	W	#	I	Measurement / Question	^ Finding	D Note
CRITICALITY	3.1	0.450	25	1	Are goto, continue, or other control flow jumps present within the code?	No	0
			26	1	Is direct or indirect recursion utilized within the code?	Few	0
			27	1	Do loops always have a fixed upper bound?	Most	0
			28	1	Is dynamic heap memory allocated after initialization?	Many	0
			29	1	Are there functions longer than 60 lines?	Many	0
			30	1	Does the assertion density average to minimally two assertions per function?	No	0
			31	1	Assertions per KSLOC of source code	0.3333	0
			32	1	Are assertions free of side effects?	Yes	0
			33	1	Do assertions have failure handling?	No	0
			34	1	Are data objects declared near their first use?	Few	0
			35	1	Is preprocessor used beyond includes, guards, and simple macros?	Few	0
			36	1	Is there evidence that the code compiles with zero (0) warnings?	No	0
			37	1	Are function return values checked consistently?	Few	0
			38	1	Does the code utilize multi-level pointer dereferences?	No	0
			39	1	Does the code utilize function pointers?	No	0
			40	1	Are data objects declared at the smallest possible level of scope?	Many	0

CREW Tab: Standards³ (Language – C/C++)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
LANGUAGE C/C++	5.0	0.450	41	1	Are goto, continue, or other control flow jumps present within the code?		Yes	0	
			42	1	Do switch statements have a default case?		No	0	
			43	1	Are case fall-throughs marked with a comment?		No	0	
			44	1	Is each statement on its own line?		No	0	
			45	1	Are variables declared on their own line?		No	0	
			46	1	Are variables declared close to their first use?		No	0	
			47	1	Are logic and assignments mixed within the same statement?		Yes	0	
			48	1	Does the code utilize the volatile keyword without a comment documenting why?		Yes	0	
			49	1	Does the code utilize void* without a comment documenting why?		Yes	0	
			50	1	Are floating point values compared to integers?		Yes	0	

CREW Tab: Standards⁴ (Language – C++)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
LANGUAGE C++			51	☐	Does the code follow the "rule of 3/5" while maintaining uniform scoping?	C	Yes	0	
			52	☐	Is the implementation of assignment/move operators/constructors safe and within accepted C++ guidelines?	C	Yes	0	
			53	☐	Are special member functions explicitly declared at the proper scope when required?	C	Yes	0	
			54	☐	Are structs used only as data containers and not as substitutes for classes?	C	Yes	0	
			55	☐	Are identifiers declared such that none are inadvertently hidden?	C	Yes	0	
			56	☐	Are identifiers overloaded with the proper keywords as required?	C	Yes	0	
			57	☐	Are virtual functions typed and declared only where circumstances allow?	C	Yes	0	
			58	☐	Are C++ keywords and symbols used properly?	C	Yes	0	
			59	☐	Do constructors use initializing methods for their class fields using best practices?	C	Yes	0	
			60	☐	Do constructors use keywords, overload appropriately, and call other constructors in the inheritance chain?	C	Yes	0	
			61	☐	Do constructor bodies use allowed functions/variables and initialize the necessary member attributes?	C	Yes	0	
			62	☐	Are destructors not called directly and only use identifiers which are safe?	C	Yes	0	
			63	☐	Is multiple inheritance used minimally and cautiously?	C	Yes	0	
			64	☐	Are classes within inheritance chains declared virtual as necessary?	C	Yes	0	
			65	☐	Are inheritance chains built properly?	C	Yes	0	
			66	☐	Are the subscript, unary, and comma operators overloaded minimally and declared appropriately?	C	Yes	0	
			67	☐	Are the comparison operators overloaded minimally, declared appropriately, and in terms of each other?	C	Yes	0	
			68	☐	Are all required variants defined when the new and delete operators are overloaded?	C	Yes	0	
			69	☐	Are the arithmetic operators overloaded minimally, declared appropriately, and in terms of each other?	C	Yes	0	
			70	☐	Are operators overloaded minimally, returning data as expected, and in terms of other operators?	C	Yes	0	
			71	☐	Is the best available smart pointer and instantiation method used and preferred over raw pointers?	C	Yes	0	
			72	☐	Are smart pointers initialized only with allowed objects or types?	C	Yes	0	
			73	☐	Are smart pointers used as function parameters minimally, following best practices?	C	Yes	0	
			74	☐	Are identifiers not reused within the same namespace?	C	Yes	0	
			75	☐	Is the global namespace referenced correctly and does it contain only the minimum needed declarations?	C	Yes	0	
			76	☐	Are unnamed namespaces and use of the 'using' keyword not utilized in header files?	C	Yes	0	
			77	☐	Are identifiers' definitions/types/declarations separated by namespaces without overlap?	C	Yes	0	
			78	☐	Are namespaces used and nested minimally, with the standard namespaces remaining unmodified?	C	Yes	0	
			79	☐	Are streams utilized in a safe manner?	C	Yes	0	
			80	☐	Does the code follow best practice guidance when using the STL?	C	Yes	0	
			81	☐	Are the member states of classes and functions immutable by outside entities?	C	Yes	0	
			82	☐	Are global identifiers encapsulated?	C	Yes	0	
			83	☐	Are templates used minimally and in a safe manner following best practices?	C	Yes	0	
			84	☐	Are exceptions, throw statements, and their handlers defined and used properly?	C	Yes	0	
			85	☐	Are exceptions and their definitions/types used in accordance with their specification?	C	Yes	0	
			86	☐	Are handlers provided for possible exceptions thrown, in order, and called only during normal program flow?	C	Yes	0	
			87	☐	Does the code only throw minimally, where allowed?	C	Yes	0	
			88	☐	Does the code follow a consistent throw/catch paradigm?	C	Yes	0	

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Maintainability - Comments

Alan Cohn

What to Look For in Maintainability - Comments

- **Quantity** - High level of commenting excluding boilerplates
 - What is the comment density and type?
 - What is the makeup of file/module headers?
 - Are prolog headers used prior to functions and classes?
- **Quality** - High quality of commenting that clarifies the code's intent
 - Does the code provide additional clarity to the code?
 - Are the comments accurate?
 - Are the comments (both content and style) consistent?

CREW Tab: Maintainability - Comments (Quantity)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
QUANTITY	1.7	0.500	1	0	Total number of source files	•	800
			2	0	Total number of (all) lines in the code base	•	350000
			3	0	Source lines of code ignoring blank and comment lines (PSLOC)	•	150000
			4	0	Number of comment lines in the code base	•	120000
			5	1	Uncorrected comment to code ratio	•	75%
			6	1	Estimated percent of source files containing a module header block		90%
			7	1	Are existing module header comment blocks merely copyright info?		No
			8	0	Approximate number of comment lines per existing module header block		50
			9	1	Estimated percent of data declarations that are commented		50%
			10	1	Estimated percent of class, function, and method declarations that are commented		50%
			11	1	Prevalence of commented-out code (which count as comment lines)		None
			12	1	Estimated total number of commented-out lines of code	•	200
			13	1	Corrected comment to code ratio	•	56%
			14	1	Overall the quantity of code comments is		Appropriate

CREW Tab: Maintainability – Comments² (Quality)

QUALITY	2.2	0.500	15	1	Estimated percent of source files with a meaningful preamble comment block re: purpose	75%
			16	1	Are code modules separated into standardized sections via comments?	All
			17	1	Are C-style /* ... */ comments used?	Yes
			18	1	Are // comments used?	Few
			19	1	Are end-of-line comments used?	Some
			20	1	Are block comments spanning multiple lines used?	Yes
			21	1	Are comments written at the level of the code's intent, i.e. what and why, not how?	Yes
			22	1	Are comments used to mark requirements being satisfied in code?	No
			23	1	Are comments used to mark developer changes or modification dates?	Yes
			24	1	Do comments exist that relate to constructs from an existing SAS or SDS?	Some
			25	1	Number of lines containing suspicious comments (todo, hack, fixme, kludge, later, bug, etc.)	100
			26	1	Estimated percent of comments that are extraction formatted (Doxygen, JavaDoc, etc.)	0%
			27	0	Are extraction formatted (Doxygen, etc.) comments used fully and consistently?	C Yes
			28	1	Are comments accurate to the code?	Yes
			29	1	Are comments written at a level to be useful for someone unfamiliar with the code?	Most
			30	1	Is the commenting style, level of detail, and formatting consistent throughout?	Yes
			31	1	Are comments indented the same as the corresponding code?	Most
			32	1	Are comments describing subsequent code lines preceded with (at least) one blank line?	Most
			33	1	Do comments add to or detract from the code's overall understandability?	Add
			34	1	Does the commenting style appear easy to maintain?	Yes
			35	0	Is any portion of the code auto-generated via a code generation tool?	No
			36	0	Is auto-generated code consistently identifiable via a comment signature?	C Yes
RISK	1.96					

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Maintainability - Clarity

Jay Marchetti

What to Look For in Maintainability - Clarity

- **Consistency** - High level of code stylistic consistency (naming, source layout, brace usage, etc)
 - Is the code carefully- and well-crafted and remain consistently so throughout the codebase?
- **Constants** - High level of use of symbolic constants
 - Does the code avoid mixing constants and literals within business logic?
- **Files / Folders** - High level of source file / folder naming consistency and organization
 - Are files and folders utilized in a way that reinforces a coherent architectural understanding?
- **Readability** - High level of overall code readability
 - Is the code layout, structure, and execution flow easily understandable?
- **Conditionals** - Low level of conditional compilation
 - Are conditional compilation directives within the code used sparingly / judiciously?

CREW Tab: Maintainability – Clarity (Consistency, Constants, Files / Folders)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
CONSISTENCY	1.4	0.200	1	1	Does the overall visual layout clarify the logical structure of the code?		Yes
			2	1	Does the code module layout lend clarity and remain consistent throughout the codebase?		Yes
			3	1	Does each line contain only one statement or data declaration?		Usually
			4	1	Is source code indented using only space (and not tab) characters?		Usually
			5	1	Is the code indentation in consistent integer multiples, e.g. multiples of 3 or 4?		Yes
			6	1	Is whitespace used consistently and in a way that clarifies / organizes the code?		Yes
			7	1	Estimated total number of commented-out lines of code	•	200
			8	1	Does the name of a type, enum, constant, variable, or routine consistently indicate which one it is?		Usually
			9	1	Are variables consistently named to accurately indicate their purpose?		Yes
			10	1	Do function or method names indicate precisely what they do?		Yes
			11	1	Do functions or methods perform one well-defined purpose?		Yes
			12	1	Are braces used consistently to enclose all decision and loop code blocks?		Usually
			13	1	Is opening and closing brace placement consistent throughout?		Usually
			14	1	Are parentheses used consistently to ensure unambiguous intent?		Yes
CONSTANTS	2.7	0.200	15	1	Are well-named constants used instead of magic numbers or magic strings?		Usually
			16	1	Estimated number of embedded numeric literals excluding the values {+ and -} 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16		600
			17	1	Are constants named for easy identification from variables, e.g. ALL_CAPS?		Most
			18	1	Are there instances of hard-coded paths or filenames within the source code?		No
			19	1	Are there instances of hard-coded ports, addresses, or other resources in the source code?		No
			20	1	Are global constants defined within a single file?		No
			21	1	Are language-standard fixed-width integer types used for integer constant declarations?		Some
			22	1	Are floating point values tested for equality or inequality?		Yes
			23	1	Is const used consistently rather than #define when constants are defined?		Some
			24	1	Is there one class, or collection of routines with one and only one purpose, per file(pair) throughout?		Yes
FILES / FOLDERS	1.6	0.100	25	1	Is there one header file, identically named except for the file extension, for each implementation file?		Usually
			26	1	Do header files contain #include of other header files?		Often
			27	1	Do subfolder names map 1:1 to documented components or sub-systems?		Yes
			28	1	Do header files use preprocessor guards consistently?		Yes
			29	1	Does each source file #include the header file of the same root name?		Yes
			30	0	Does the code utilize any 3rd-party or auto-generated source code?		No
			31	0	Is all 3rd-party or auto-generated source code isolated within files and / or sub-folders?	C	Yes

Maintainability - Clarity

- As usual with CREW, read over a random subset of the source code before answering any of the questions.
- Read code with this mindset: “What if I had to maintain this code?”

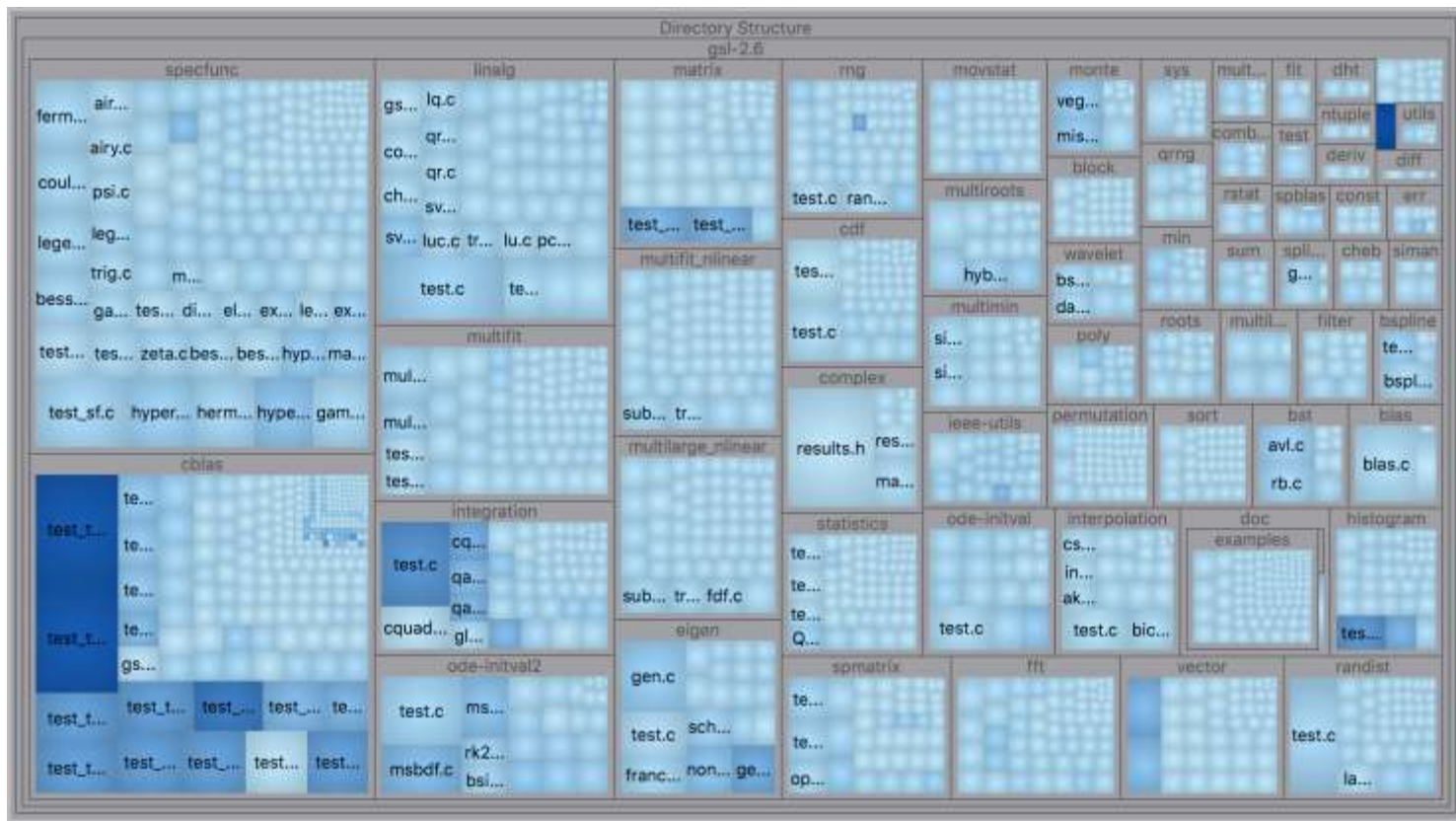
TIP: Also use SA tool checkers, metrics, and visualizations to focus on targeted areas for code reading.

Cyclomatic Complexity – Codebase Hot-spot Visualization

A **Treemap** visualization shows all files in the code base and their relative CC.

Here cell size indicates each file's SLOC and **blueness** indicates its maximum function CC.

Code base: GNU Scientific Library



Treemap Created Using SciTools Understand

CREW Tab: Maintainability – Clarity² (Readability)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
READABILITY	3.2	0.300	32	1	Total number of source files	•	800
			33	1	Number of definition (header) files	•	300
			34	1	Number of implementation (code) files	•	500
			35	1	Total number of (all) lines in the code base	•	350000
			36	1	Source lines of code ignoring blank and comment lines (PSLOC)	•	150000
			37	1	Executable source lines of code (ESLOC)	•	100000
			38	1	Total number of functions and methods within the code base	•	5000
			39	1	Average lines of code per file (PSLOC / total # of source files)	•	188
			40	1	Average lines of code per function or method (PSLOC / # functions and methods)	•	30
			41	1	Average number of functions or methods implemented per file (# functions and methods / # of impl. files)	•	10.0
			42	1	Percentage of functions or methods exceeding 200 lines of code in length		1.0%
			43	1	Percentage of source files that exceed 1000 lines of code in length?		2.0%
			44	1	Average file cyclomatic complexity (CC1)	•	3.333
			45	1	Maximum cyclomatic complexity (CC1) for any function within the code base	•	100
			46	1	Percentage of functions or methods within the code exceeding cyclomatic complexity (CC1) of 20		1.0%
			47	1	Percentage of functions or methods within the code exceeding strict cyclomatic complexity (CC2) of 20		0.8%
			48	1	Maximum control structure nesting level within the code	•	8
			49	1	Cyclomatic complexity (total CC1) of the application v(G)	•	20000
			50	1	Cyclomatic density (total CC1 / ESLOC)	•	0.20
			51	1	Structuredness, i.e. essential complexity (total) of the application Ev(G)	•	5000
			52	1	Essential density (total essential complexity / total cyclomatic complexity (CC1))	•	0.56
			53	1	Percentage of implementation files with normalized Maintainability Index (MI3) in 'difficult' range (< 38)		15.0%
			54	1	Percentage of implementation files with normalized Maintainability Index (MI3) in 'moderate' range (>=38 MI <=50)		45.0%
			55	1	Percentage of implementation files with normalized Maintainability Index (MI3) in 'high' range (>50)	•	40.0%
			56	1	Do source files begin with a meaningful module header block regarding their purpose?		No
			57	1	Are files separated into consistent sections, e.g. file preamble, includes, defines, data types, functions, etc?		Yes
			58	1	Are methods or functions within a file separated with a blank line?		Yes
			59	1	Is the nominal path through the code generally clear to the reader?		Yes
			60	1	Does the code use simple constructs rather than abstruse language features?		Usually
			61	1	Does the code avoid complex multiple conditions within decisions?		Usually
			62	1	Do loops and functions have just one exit path?		Sometimes
			63	1	Does the code make use of 'goto' to modify the execution path?		Sometimes
			64	1	Number of occurrences of direct or indirect recursion within the code		4
			65	1	Are functions implementing tasks or threads named in a way to make this clear? E.g. ending in "_task"?		Yes

Maintainability - Clarity

- Note that a number of the Readability questions are answered Automatically as indicated by the "A" column.

CREW Tab: Maintainability – Clarity³ (Conditionals)

Focus	Risk	W	#	I	Measurement / Question	Finding
CONDITIONALS	4.0	0.200	66	1	Number of unique conditional compile flags used within the code	50
			67	1	Are conditionals prevalent enough within the code to hinder understanding it?	Somewhat
			68	1	Is there a mechanism to indicate at run-time which conditionals were used at build-time?	No
RISK	2.73					

Maintainability - Clarity

- Too many conditional compilations impede the code's understandability and executable's precise source code provenance.
- We use an SEI script to determine the number of unique conditionals.

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Maintainability - Reusability

Alan Cohn

What to Look For in Maintainability - Reusability

- **Independence** - High level of machine, OS, database, and compiler independence
 - Is the code portable to many different platforms?
 - Are types mixed throughout the code?
- **Utilization** - Low level of commented-out or dead code, and unused variables
 - Does the code have unused variables or functions?
 - Is commented or no-effect code present?
- **Complication** - Low level of complicated implementation constructs
 - What is the maximum control nesting structure?
 - How is the distribution of KSF in code?
- **Duplication** - Low level of duplicated (copy / pasted) code
 - How much duplication is there?

CREW Tab: Maintainability - Reusability (Independence)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
INDEPENDENCE	2.3	0.250	1	1	Does the code utilize in-line assembly language, e.g. the "asm" keyword?		No
			2	1	Does the code use other machine dependent constructs, e.g. "register" variables?		Often
			3	1	Does the code contain constructs that implicitly assume a particular data model (i.e. data type size) such as ILP32?		No
			4	1	Does the code make assignments between integers, longs, and pointers?		No
			5	1	Does the code mix signed and unsigned integers within expressions?		No
			6	1	Does the code make use of bitfields?		Often
			7	1	Does the code use subsystems to hide implementation details for OS, database, BSP, or device access?		No
			8	1	Are there time delays within the code that are built with simple for or while loop counters?		No
			9	1	Does the code utilize global variables within interrupt routines or by multiple threads?		No
			10	1	Does the code contain constructs that implicitly assume big or little endianness?		No
			11	1	Does the code only utilize built-in platform-specific compiler flags, e.g. #ifdef _WIN32 rather than custom ones?		Yes
			12	1	Are system calls embedded directly within the code?		Sometimes
			13	1	Does the code utilize constants instead of sizeof() to encode storage sizes within expressions?		Yes
			14	0	Does the code utilize any 3rd-party or auto-generated source code?		No
			15	0	Is 3rd-party or auto-generated code intermixed along with custom code within source files?	C	No
			16	0	Is 3rd-party or auto-generated code readily identifiable as such (whether in intermixed files or isolated files / sub-folders)?	C	Sometimes

CREW Tab: Maintainability – Reusability² (Utilization)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
UTILIZATION	3.4	0.250	17	1	Estimated total number of commented-out lines of code	•	200
			18	1	Are commented-out lines interspersed freely among uncommented-out lines of code?		Sometimes
			19	1	Estimated total number of unreachable statements within the code		20
			20	1	Estimated total number of unused functions or methods within the code		10
			21	1	Estimated total number of unused global variables within the code		10
			22	1	Estimated total number of unused local variables within the code		10
			23	1	Estimated total number of no-effect or redundant statements within the code		10

CREW Tab: Maintainability – Reusability³ (Complication)

Focus	Risk	W	#	1	Measurement / Question	▲	Finding
COMPLICATION	4.0	0.250	24	1	Maximum control structure nesting level within the code	•	8
			25	0	Total number of functions or methods within the code base	•	5000
			26	1	Percentage of functions or methods within the code with Hersteller (HIS) VOCF (LanguageScope) exceeding 4.0		10.0%
			27	1	Number of goto occurrences within the code, i.e. Hersteller GOTO metric		10
			28	1	Number of functions within the code with Hersteller AP_CG_CYCLE (Recursion) of 1 or more		5
			29	1	Percentage of functions or methods within the code with Hersteller CALLING (FanIn) exceeding 5		10.0%
			30	1	Percentage of functions or methods within the code with Hersteller CALLS (FanOut) exceeding 7		10.0%
			31	1	Percentage of functions or methods within the code with Hersteller PARAM (NumParams) exceeding 5		10.0%
			32	1	Percentage of functions or methods within the code with Hersteller LEVEL (call nesting) exceeding 4		10.0%
			33	1	Average Koopman Spaghetti Factor (KSF) of all implementation files		5.0
			34	1	Percentage of implementation files with KSF exceeding 15		5.0%

CREW Tab: Maintainability – Reusability⁴ (Duplication)

Focus	Risk	W	#	I	Measurement / Question	A	Finding
DUPLICATION	5.0	0.250	35	1	What is the level of duplicated code lines, i.e. how much copy / pasted code is there?		Much
			36	1	Estimated percentage of source files (i.e. file count) that contain any duplicated code		3.0%
			37	0	Source lines of code ignoring blank and comment lines (PSLOC)	•	150000
			38	1	Total number of lines of duplicated code detected		36744
			39	1	Duplication ratio (as a percentage) = total number of detected duplicated lines to PSLOC	•	24.5%

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Fault Tolerance

Alan Cohn

What to Look For in Fault Tolerance

- **Value Checking** - High level of return and parameter value checking
 - Does the code consistently check for / return proper input and output values?
- **Error Handling** - Existence of a systematic error handling mechanism
 - Does the code utilize a consistent means of managing errors and exceptions?
- **Error Logging** - Existence of an error / fault logging mechanism
 - Are error and exception information logged sufficiently and consistently?
- **Instrumentation** - Existence of run-time statistics and timing instrumentation
 - Does the code monitor and record average and worst-case run-time characteristics?
- **Health Monitoring** - Effective use of watchdog and system health monitoring
 - Does the code monitor the system and attempt to restart the system if necessary?

CREW Tab – Fault Tolerance

Focus	Risk	W	#	1	Measurement / Question	A	Finding	D
VALUE CHECKING	3.8	0.300	1	1	Are function return values checked consistently?		Few	1
			2	1	Estimated number of calls without the return value checked		100	1
			3	1	Are there instances of unchecked return values with subsequent usage?		Many	1
			4	1	Are there instances where all possible return values from a function are not accounted for?		Yes	1
			5	1	Are there instances of functions not returning a value or throwing an exception on an error?		Somewhat	1
			6	1	Are pointers confirmed to be correctly initialized and non-null before use?		Minimal	1
			7	1	Are parameter values within functions confirmed to be within expected bounds before use?		Many	1
			8	1	Are user inputs confirmed to be of the correct type?		Somewhat	1
			9	1	Are user inputs confirmed to be within an expected range?		Minimally	1
ERROR HANDLING	4.7	0.300	10	1	Is there evidence of a systematic error handling mechanism?		Minimal	1
			11	1	Do functions / methods consistently return an error or throw an exception on failure?		No	1
ERROR LOGGING	3.0	0.200	12	1	Are errors consistently logged?		Somewhat	1
			13	1	Are exceptional conditions consistently logged?		Mostly	1
			14	1	Are security events consistently logged?		No	1
			15	1	Are logs sufficiently detailed to identify the issue?		Mostly	1
			16	1	Are stored logs maintained to not exceed a maximum allowable size?		No	1
INSTRUMENTATION	3.8	0.100	17	1	Are worst-case and average run-time processor utilization recorded?		No	1
			18	1	Are run-time communication channels utilization recorded?		Yes	1
			19	1	Are worst-case and average run-time memory utilization recorded?		No	1
			20	1	Are worst-case and average run-time timing latencies recorded?		No	1
			21	1	Is boot-up (turn on) counts recorded?		Yes	1
			22	1	Is system (total) run-time hours recorded?		No	1
HEALTH MONITORING	2.9	0.100	23	1	Does the system utilize a watchdog timer?		Yes	1
			24	1	Does the code verify the health of all tasks and threads before strobing the watchdog?		No	1
			25	1	Are fault and watchdog reboot counts recorded?		No	1
			26	1	Are system health and instrumentation data accessible to system administrators?		Yes	1
RISK	3.81							

CREW Tab – Fault Tolerance – Value Checking

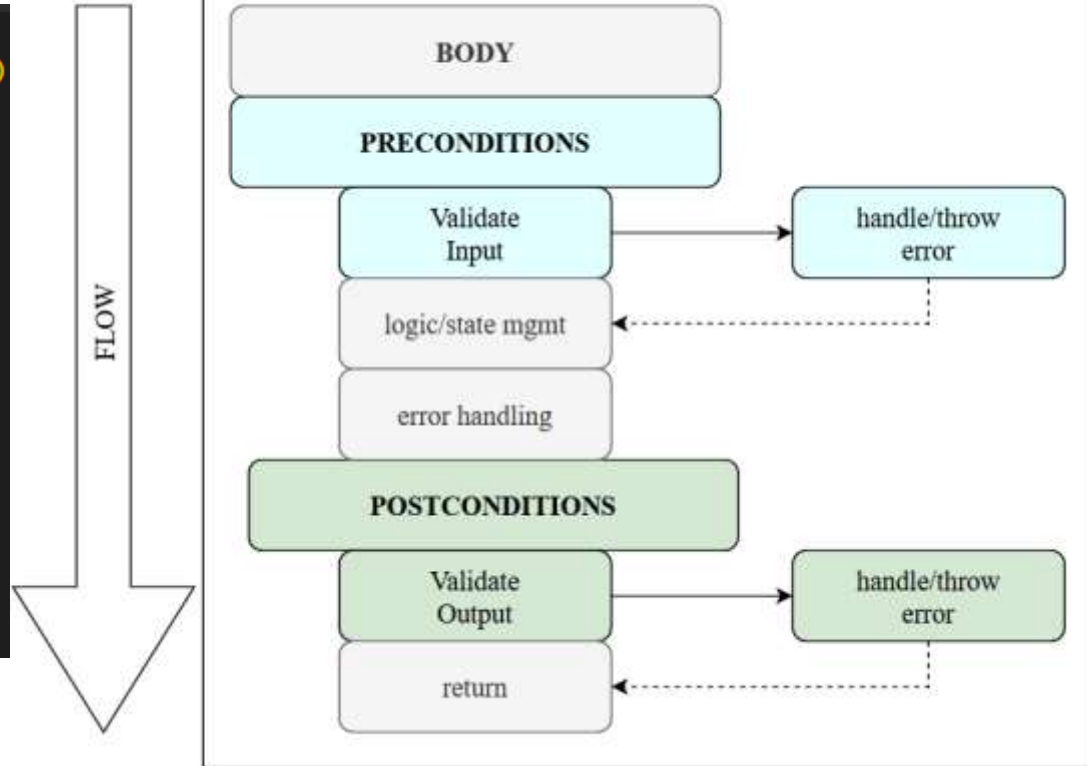
```
2 int saveRecords( void ** pData, int szData )
3 {
4     int recordsSaved, idx;
5
6     MyRecord ** pRecords = ( MyRecord ** )pData;           // #8
7
8     while ( idx < szData &&                                // #7, 9
9           pRecords[ szData - ( idx + 1 ) ]->isValid() ) // #6, 7, 9
10     {
11         MyParsedRecord * pParsed =
12         parse( pRecords[ szData - ( idx + 1 ) ] );           // #1 [user's return]
13
14         E_RecordState recordState = pParsed->getState();      // #3
15
16         switch ( recordState )
17         {
18             case E_RecordState::error:
19                 throw MyException( "invalid state!" );        // #5 [throw error]
20
21             case E_RecordState::incomplete:
22                 return;                                       // #5 [no value]
23         }
24
25         bool isSaved = saveRecord( pParsed );                 // #1 [user's return]
26
27         ++recordsSaved;
28         ++idx;
29     }
30
31     return recordsSaved;                                     // #1 [our return]
32 }
```

VALUE CHECKING		1.8	0.300	1	1	Are function return values checked consistently?
				2	1	Estimated number of calls without the return value checked
				3	1	Are there instances of unchecked return values with subsequent usage?
				4	1	Are there instances where all possible return values from a function are not accounted for?
				5	1	Are there instances of functions not returning a value or throwing an exception on an error?
				6	1	Are pointers confirmed to be correctly initialized and non-null before use?
				7	1	Are parameter values within functions confirmed to be within expected bounds before use?
				8	1	Are user inputs confirmed to be of the correct type?
				9	1	Are user inputs confirmed to be within an expected range?

1	1	Are function return values checked consistently?
2	1	Estimated number of calls without the return value checked
3	1	Are there instances of unchecked return values with subsequent usage?
4	1	Are there instances where all possible return values from a function are not accounted for?
5	1	Are there instances of functions not returning a value or throwing an exception on an error?
6	1	Are pointers confirmed to be correctly initialized and non-null before use?
7	1	Are parameter values within functions confirmed to be within expected bounds before use?
8	1	Are user inputs confirmed to be of the correct type?
9	1	Are user inputs confirmed to be within an expected range?

CREW Tab – Fault Tolerance – Value Checking

```
1
2 int saveRecords( void ** pData, int szData )
3 {
4     // PRECONDITIONS
5     assert( pData != NULL );
6     assert( szData > 0 );
7     ...
8
9     // handle errors/exceptions
10    switch ( recordState ) { ... }
11    if ( !saveRecord( pParsed ) ) { ... }
12
13    ...
14    // POSTCONDITIONS
15    assert( recordsSaved >= 0 );
16    return recordsSaved;
17 }
```



CREW Tab – Fault Tolerance – Error Handling/Logging

```
1  MyConfig * createConfig( const char * configFile,  
2  |                       | const char * securityChecksum )  
3  |                       |  
4  {  
5      FILE * pFile = fopen( configFile, "r" );  
6      if ( pFile == NULL )  
7          return NULL; // #12, 13  
8  
9      MyConfig * pConfig = deserialize( pFile );  
10     if ( pConfig == NULL )  
11     {  
12         LOG_ERROR( "unable to deserialize" ); // #15  
13         throw MyException( "unable to create config" ); // #10, 11  
14     }  
15     // has config been tampered with?  
16     const char * checksum = calcChecksum( pConfig );  
17     if ( strcmp( checksum, securityChecksum ) != 0 )  
18         return NULL; // #12-14  
19  
20     return pConfig;  
21 }
```

ERROR HANDLING	4.7	0.300	10	1	Is there evidence of a systematic error handling mechanism?
			11	1	Do functions / methods consistently return an error or throw an exception on failure?
ERROR LOGGING	3.0	0.200	12	1	Are errors consistently logged?
			13	1	Are exceptional conditions consistently logged?
			14	1	Are security events consistently logged?
			15	1	Are logs sufficiently detailed to identify the issue?
			16	1	Are stored logs maintained to not exceed a maximum allowable size?

10	1	Is there evidence of a systematic error handling mechanism?
11	1	Do functions / methods consistently return an error or throw an exception on failure?
12	1	Are errors consistently logged?
13	1	Are exceptional conditions consistently logged?
14	1	Are security events consistently logged?
15	1	Are logs sufficiently detailed to identify the issue?
16	1	Are stored logs maintained to not exceed a maximum allowable size?

CREW Tab – Fault Tolerance – Monitoring / Runtime



INSTRUMENTATION	3.8	0.100	17	1	Are worst-case and average run-time processor utilization recorded?
			18	1	Are run-time communication channels utilization recorded?
			19	1	Are worst-case and average run-time memory utilization recorded?
			20	1	Are worst-case and average run-time timing latencies recorded?
			21	1	Is boot-up (turn on) counts recorded?
			22	1	Is system (total) run-time hours recorded?
			23	1	Does the system utilize a watchdog timer?
HEALTH MONITORING	2.9	0.100	24	1	Does the code verify the health of all tasks and threads before strobing the watchdog?
			25	1	Are fault and watchdog reboot counts recorded?
			26	1	Are system health and instrumentation data accessible to system administrators?

Instrumentation measures:

- resource utilization
- performance / runtime latencies
- restarts / runtime

Health Monitoring measures:

- process states
- process hangs / crashes
- system restarts

17	1	Are worst-case and average run-time processor utilization recorded?
18	1	Are run-time communication channels utilization recorded?
19	1	Are worst-case and average run-time memory utilization recorded?
20	1	Are worst-case and average run-time timing latencies recorded?
21	1	Is boot-up (turn on) counts recorded?
22	1	Is system (total) run-time hours recorded?
23	1	Does the system utilize a watchdog timer?
24	1	Does the code verify the health of all tasks and threads before strobing the watchdog?
25	1	Are fault and watchdog reboot counts recorded?
26	1	Are system health and instrumentation data accessible to system administrators?

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Testability

Alan Cohn

What to Look For in Testability

- **Unit** - Evidence of unit testing, especially automated (test cases, test harness, etc.)
 - Does the code show utilization of a test harness and test cases for unit testing?
 - Are the test cases written to go beyond happy paths and test edge cases?
 - Do the unit tests provide ample code coverage?
- **Regression** - Evidence of regression testing, especially automated
 - Does the code or code commenting indicate the use of regression testing?
- **Complexity** - Low valued cyclomatic complexity and path count
 - Is function / method code complexity consistently low enough to allow thorough testing?

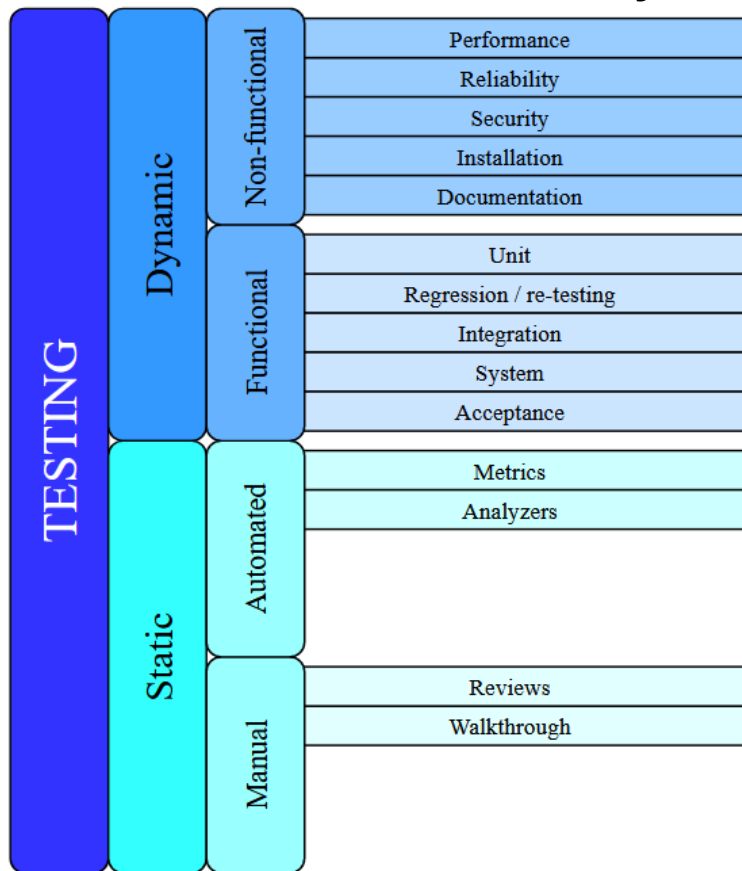
Testability – SEI's Perspective

One of the most consistently overlooked and underutilized methods of reducing defects, preventing side-effects, and establishing confidence in the product.

CREW's Three Main Areas Distilled:

1. How do we prevent unexpected side-effects after making changes anywhere in the code base?
2. How can we verify consistent behavior across time, usage, expectations, and unexpected circumstances?
3. How do we prevent complexity from becoming unmanageable?

CREW Tab – Testability - Overview



Unit Testing

What:

Tests smallest components of code

Why:

Ensures expected behavior of components

Regression Testing

What:

Revalidates parts of or entire system by running test data against it

Why:

Prevents new bugs from being introduced due to modifications

Complexity Metrics

What:

Helps indicates cognitive load on programmer

Why:

Reduces maintenance times and testability

CREW Tab – Testability CREW Worksheet

Focus	W	#	I	Measurement / Question
UNIT	2.6	0.350	1	1 Does the supplied code base include the unit test code?
			2	1 Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?
			3	1 Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?
			4	1 Do unit tests utilize assertions liberally to indicate test success or failure?
			5	1 Does the coverage of the unit tests appear to be comprehensive?
			6	1 Does the code base utilize global variables?
			7	1 Does the test code allow easy documentation of application or library code?
			8	1 Do tests utilize both positive and negative test cases, i.e. fault injection?
			9	1 Are the unit tests generally easy to understand and verifiable?
REGRESSION	4.0	0.300	10	1 Is there evidence in either code or comments that the code is regression tested?
			11	1 Is there evidence of test driver usage or other automated regression testing?
			12	1 Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?
COMPLEXITY	3.9	0.350	13	1 Maximum cyclomatic complexity (CC1) for any function within the code base
			14	0 Total number of functions or methods within the code base
			15	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10
			16	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20
			17	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30
			18	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40
			19	1 Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50
			20	1 Number of functions within the code with cyclomatic complexity (CC1) > 50
			21	1 Average file cyclomatic complexity (CC1)
			22	1 Average file strict cyclomatic complexity (CC2)
			23	1 Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10
			24	1 Average file modified cyclomatic complexity (CC3)
			25	1 Percentage of functions or methods within the code with Hersteller PATH exceeding 80
RISK	3.44			

CREW Tab – Testability – Unit Testing

UNIT	2.6	0.350	1	1	Does the supplied code base include the unit test code?
			2	1	Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?
			3	1	Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?
			4	1	Do unit tests utilize assertions liberally to indicate test success or failure?
			5	1	Does the coverage of the unit tests appear to be comprehensive?
			6	1	Does the code base utilize global variables?
			7	1	Does the test code allow easy documentation of application or library code?
			8	1	Do tests utilize both positive and negative test cases, i.e. fault injection?
			9	1	Are the unit tests generally easy to understand and verifiable?

1	1	Does the supplied code base include the unit test code?
2	1	Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?
3	1	Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?
4	1	Do unit tests utilize assertions liberally to indicate test success or failure?
5	1	Does the coverage of the unit tests appear to be comprehensive?
6	1	Does the code base utilize global variables?
7	1	Does the test code allow easy documentation of application or library code?
8	1	Do tests utilize both positive and negative test cases, i.e. fault injection?
9	1	Are the unit tests generally easy to understand and verifiable?

CREW Tab – Testability – Regression Testing

REGRESSION	4.0	0.300	10	1	Is there evidence in either code or comments that the code is regression tested?
			11	1	Is there evidence of test driver usage or other automated regression testing?
			12	1	Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?

Is there evidence in either code or comments that the code is regression tested?

Is there evidence of test driver usage or other automated regression testing?

Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?

CREW Tab – Testability – Complexity

COMPLEXITY	3.9	0.350	13	1	Maximum cyclomatic complexity (CC1) for any function within the code base
			14	0	Total number of functions or methods within the code base
			15	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10
			16	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20
			17	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30
			18	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40
			19	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50
			20	1	Number of functions within the code with cyclomatic complexity (CC1) > 50
			21	1	Average file cyclomatic complexity (CC1)
			22	1	Average file strict cyclomatic complexity (CC2)
			23	1	Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10
			24	1	Average file modified cyclomatic complexity (CC3)
			25	1	Percentage of functions or methods within the code with Hersteller PATH exceeding 80

13	1	Maximum cyclomatic complexity (CC1) for any function within the code base
14	0	Total number of functions or methods within the code base
15	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10
16	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20
17	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30
18	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40
19	1	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50
20	1	Number of functions within the code with cyclomatic complexity (CC1) > 50
21	1	Average file cyclomatic complexity (CC1)
22	1	Average file strict cyclomatic complexity (CC2)
23	1	Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10
24	1	Average file modified cyclomatic complexity (CC3)
25	1	Percentage of functions or methods within the code with Hersteller PATH exceeding 80

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Security - General

Alan Cohn

What to Look For in Security - General

- **Defensiveness** - High level of defensive / secure coding practices
- **Validation** - High level of validation of untrusted inputs / interfaces
- **Libraries** - Low level of use of potentially unsafe library calls
- **Warnings** - Evidence that code compiles with no (or few) warnings
- **Traceability** - Effective use of executable and parameter file signatures / anti-tamper measures

Analyzing Security

- Some questions, predominantly in Defensiveness and Libraries can be answered very simply with SA tools.
- Most rely more heavily on manual code review.
- Start by finding evidence of encryption.
- Once you know the functions used for encryption, you can use your SA tools or IDE to find instances of its use to help answer the CREW questions.
- Note: Warnings requires build output to answer. If you are not personally building the code, make sure to have active communications with the person building the project early in the analysis process.

CREW Tab: Security - General (Defensiveness, Validation)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
DEFENSIVENESS	3.6	0.200	1	1	Are magic numbers excluding the values (+ and -) 0, 0.0, 1, 1.0, 2, 4, 8, 10, 16 prevalent in the code?		Many	0	
			2	1	Are expressions within macros fully parenthesized?		Many	0	
			3	1	Are numbers defined by the preprocessor parenthesized?		No	0	
			4	1	Are macros reserved for simple highly reusable functions?		Somewhat	0	
			5	1	Are variables checked before used in operations that may cause overflows or divide by zero errors?		Many	0	
			6	1	Are pointers confirmed to be correctly initialized and non-null before use?		Somewhat	0	
			7	1	Are parameters within functions confirmed to be within expected bounds before use?		No	0	
			8	1	Are return values checked for errors?		No	0	
			9	1	Is there evidence of a unifying coding standard?		Somewhat	0	
			10	1	Are exceptional conditions handled consistently?		Minimal	0	
			11	1	Does the code exhibit graceful degradation?		No	0	
			12	1	Is trusted and untrusted data stored within the same data structure?		Yes	0	
VALIDATION	3.8	0.200	13	1	Is hashing used to verify data sources?		No	0	
			14	1	Are modern hashing algorithms used?		No	0	
			15	1	Are encryption algorithms used when passing data through untrusted channels?		No	0	
			16	1	Is sensitive data at rest encrypted?		None	0	
			17	1	Are modern encryption algorithms used?		No	0	
			18	1	Is memory cleaned between reads?		No	0	
			19	1	Is message acknowledgment utilized?		Yes	0	
			20	1	Are interrupts used over polling for communication buffers?		No	0	
			21	1	Does privileged activity undergo secondary authentication?		No	0	
			22	1	Is all user input confirmed to be of the correct type?		Yes	0	
			23	1	Is all user input confirmed to be within an expected range?		Yes	0	
			24	1	Is all user input sanitized?		Yes	0	

CREW Tab: Security² - General (Libraries, Warnings, Traceability)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
LIBRARIES	3.0	0.200	25	1	Is a real-time OS used?		No	0	
			26	1	Are any library calls or run-time API's with known security concerns utilized?		No	0	
			27	1	Are any deprecated library functions used?		Yes	0	
			28	1	Are returned values from library functions consistently checked and verified?		Yes	0	
			29	0	Does the code base utilize any 3rd-party source code?		No	0	
			30	0	Are the most recent versions of third-party source code used?	C	No	0	
			31	0	Is there evidence that third-party source code has been modified for system utilization?	C	Yes	0	
WARNINGS	2.3	0.200	32	1	Is there evidence that the code compiles with zero (0) warnings?		Yes	0	
			33	1	Was the code compiled as part of this code risk estimation analysis?		Yes, modified	0	
			34	1	Does the supplied makefile set warning flags to their highest level, e.g., -Wall, -Wextra, -Weverything?		Yes	0	
			35	1	Warnings count if the code is compiled with the highest warning flags set? (enter 25 if not compiled)		250	0	
			36	1	Are warning suppressions present within the code base?		Some	0	
			37	1	Does the code validate the executable image against a digital signature to ensure anti-tamper?		No	0	
TRACEABILITY	3.9	0.200	38	1	Do parameter files used by the program contain a signature that is verified at boot-time?		No	0	
			39	1	Are parameter files used by the program encrypted?		Yes	0	

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Security - CERT

Alan Cohn

What to Look For in Security - CERT

- **Density** - Low density of secure coding issues as per SEI CERT Secure Coding Rules
- **Priority** - Low number of high-priority secure coding issues as per SEI CERT Secure Coding Rules

CERT C and CERT C++ are secure coding standards developed by SEI CERT to develop safe, reliable, and secure systems.

The Security CERT & CERT Summary Tabs

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
DENSITY C/C++	2.9	0.900	1	0	Source lines of code (PSLOC) divided by 1000		150.00	0	
			2	1	Diagnostics / KSLOC Priority 1		12.97	0	
			3	1	Diagnostics / KSLOC Priority 2		0.65	0	
			4	1	Diagnostics / KSLOC Priority 3		7.18	0	
			5	1	Diagnostics / KSLOC Priority 4		3.51	0	
			6	1	Diagnostics / KSLOC Priority 6		4.83	0	
			7	1	Diagnostics / KSLOC Priority 9		16.93	0	
DENSITY C++			8	0	Diagnostics / KSLOC Priority 1	C	2.81	0	
			9	0	Diagnostics / KSLOC Priority 2	C	11.51	0	
			10	0	Diagnostics / KSLOC Priority 3	C	0.25	0	
			11	0	Diagnostics / KSLOC Priority 4	C	7.33	0	
			12	0	Diagnostics / KSLOC Priority 6	C	10.00	0	
			13	0	Diagnostics / KSLOC Priority 9	C	3.72	0	
PRIORITY C/C++	4.0	0.100	14	1	Number of Priority 6 diagnostics		724	0	
			15	1	Number of Priority 9 diagnostics		2540	0	
PRIORITY C++			16	0	Number of Priority 6 diagnostics	C	1500	0	
			17	0	Number of Priority 9 diagnostics	C	485	0	
RISK	3.03								

The SecurityCERT tab is completely auto-calculated from data in the CERTSummary tab shown below.

In the CERTSummary tab, the analyst enters the number of SA diagnostics for each of the listed CERT rules.

Category	Rule ID	Rule Description	Diagnostics Count	Diagnostics Density
Overview	Total Diagnostics		12181	86.131
	Priority 1 Diagnostics		2361	15.790
	Priority 2 Diagnostics		1645	12.187
	Priority 3 Diagnostics		1314	7.627
	Priority 4 Diagnostics		1957	13.897
	Priority 9 Diagnostics		2524	18.481
Progression C	PRC30-C	Do not create a process object before thread creation	0	0.000
	PRC31-C	Do not use a return to process to create a process	0	0.000
	PRC32-C	Do not use process object directly to create a process	140	0.007
	PRC33-C	Do not use process object directly to create a process	0	0.000
	PRC34-C	Do not use process object directly to create a process	0	0.000
	PRC35-C	Do not use process object directly to create a process	0	0.000
Declarations & Initialization C	DCI30-C	Do not declare a variable before its definition	0	0.000
	DCI31-C	Do not declare a variable before its definition	0	0.000
	DCI32-C	Do not declare a variable before its definition	0	0.000
	DCI33-C	Do not declare a variable before its definition	0	0.000
	DCI34-C	Do not declare a variable before its definition	0	0.000
	DCI35-C	Do not declare a variable before its definition	0	0.000
	DCI36-C	Do not declare a variable before its definition	0	0.000
	DCI37-C	Do not declare a variable before its definition	0	0.000
	DCI38-C	Do not declare a variable before its definition	0	0.000
	DCI39-C	Do not declare a variable before its definition	0	0.000
Expressions C	EXP30-C	Do not use a variable before its definition	0	0.000
	EXP31-C	Do not use a variable before its definition	0	0.000
	EXP32-C	Do not use a variable before its definition	0	0.000
	EXP33-C	Do not use a variable before its definition	0	0.000
	EXP34-C	Do not use a variable before its definition	0	0.000
	EXP35-C	Do not use a variable before its definition	0	0.000
	EXP36-C	Do not use a variable before its definition	0	0.000
	EXP37-C	Do not use a variable before its definition	0	0.000
	EXP38-C	Do not use a variable before its definition	0	0.000
	EXP39-C	Do not use a variable before its definition	0	0.000

- Note: Multiple SA tools should be utilized with their results aggregated. We will take more detailed look at the CERT C/C++ standards and diagnostic aggregation later in this presentation.

CREW Tab: Security – CERT (Density, Priority)

Focus	Risk	W	#	I	Measurement / Question	A	Finding	D	Note
DENSITY C/C++	2.9	0.900	1	0	Source lines of code (PSLOC) divided by 1000	•	150.00	0	
			2	1	Diagnostics / KSLOC Priority 1		12.97	0	
			3	1	Diagnostics / KSLOC Priority 2		0.65	0	
			4	1	Diagnostics / KSLOC Priority 3		7.18	0	
			5	1	Diagnostics / KSLOC Priority 4		3.51	0	
			6	1	Diagnostics / KSLOC Priority 6		4.83	0	
			7	1	Diagnostics / KSLOC Priority 9		16.93	0	
DENSITY C++			8	0	Diagnostics / KSLOC Priority 1	C	2.81	0	
			9	0	Diagnostics / KSLOC Priority 2	C	11.51	0	
			10	0	Diagnostics / KSLOC Priority 3	C	0.25	0	
			11	0	Diagnostics / KSLOC Priority 4	C	7.33	0	
			12	0	Diagnostics / KSLOC Priority 6	C	10.00	0	
			13	0	Diagnostics / KSLOC Priority 9	C	3.23	0	
PRIORITY C/C++	4.0	0.100	14	1	Number of Priority 6 diagnostics		724	0	
			15	1	Number of Priority 9 diagnostics		2540	0	
PRIORITY C++			16	0	Number of Priority 6 diagnostics	C	1500	0	
			17	0	Number of Priority 9 diagnostics	C	485	0	
RISK	3.03								

Assessing Code Quality and Security

Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Tools

Alan Cohn

CREW Tab: Tools

#	Tool Utilized	Version	Manufacturer	Notes
1	CppCheck	2.4.1	FOSS	Static analysis
2	Understand	6.0, Build 1063	SciTools	Static analysis, CodeCheck rule checker
3	Parasoft C/C++ Test	TBD	Parasoft	Static analysis, esp. security
4	FlexeLint	TBD	Gimpel	Static analysis
5	UCC	2018.07	FOSS	Static analysis
6	Lattix Architect	11.0	Lattix	Static analysis, esp. architecture
7	SCALE	TBD	SEI / FOSS	Tool diagnostics aggregation
8	Sublime Text 3	3.2.2	Sublime Text	Editor, code searches

The Tools tab captures a record of which tools and versions were used for a particular CREW analysis.

Assessing Code Quality and Security

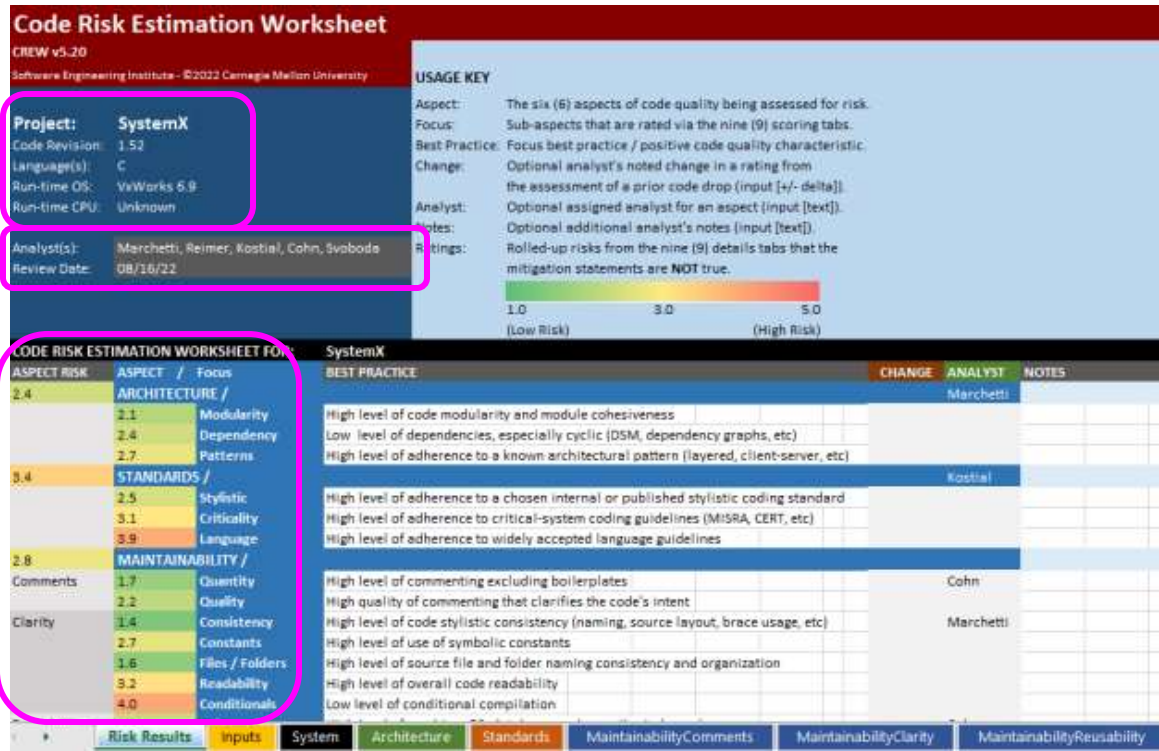
Code Risk Estimation Worksheet (CREW)

CREW Tab Walkthrough

Risk Results

Jay Marchetti

CREW Tab: Risk Results



- The Risk Results tab presents a synopsis of the code analysis, showing:
 - Basic information regarding the codebase such as the system name, revision, language, and run-time environment.
 - It also shows who performed the analysis and when.
- The Risk Results tab then displays the rolled-up risk scores for all 6 quality aspects and 32 foci.

The Risk Results tab is CREW's top-level output view for code quality assessments.

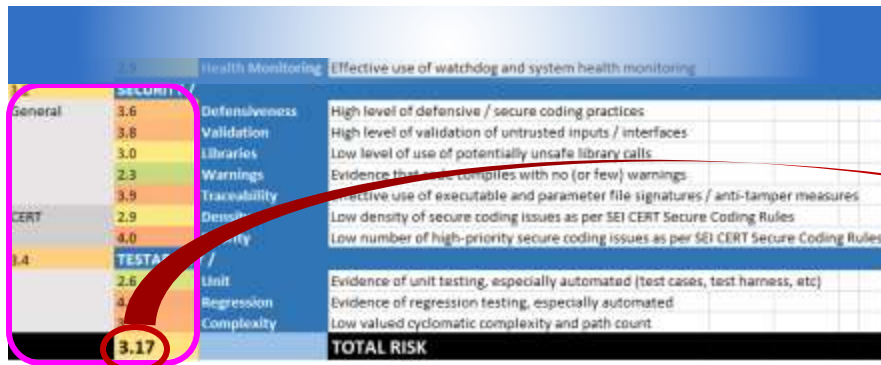
CREW Tab: Risk Results²

CODE RISK ESTIMATION WORKSHEET FOR: SystemX			CHANGE	ANALYST	NOTES
ASPECT RISK	ASPECT / Focus	BEST PRACTICE			
2.4	ARCHITECTURE /			Marchetti	
2.1	Modularity	High level of code modularity and module cohesiveness			
2.4	Dependency	Low level of dependencies, especially cyclic (DSM, dependency graphs, etc)			
2.7	Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)			
3.4	STANDARDS /			Kostial	
2.5	Stylistic	High level of adherence to a chosen internal or published stylistic coding standard			
3.1	Criticality	High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)			
3.5	Language	High level of adherence to widely accepted language guidelines			
2.8	MAINTAINABILITY /				
1.7	Quantity	High level of commenting excluding boilerplates		Cohn	
2.2	Quality	High quality of commenting that clarifies the code's intent			
1.4	Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)		Marchetti	
2.7	Constants	High level of use of symbolic constants			
1.6	Files / Folders	High level of source file and folder naming consistency and organization			
3.2	Readability	High level of overall code readability			
4.0	Conditionals	Low level of conditional compilation			
2.3	Independence	High level of machine, OS, database, and compiler independence		Cohn	
3.4	Utilization	Low level of commented-out or dead code, and unused variables			
4.0	Complexion	Low level of complicated implementation constructs			
5.0	Duplication	Low level of duplicated (copy / pasted) code			
3.8	FAULT TOLERANCE /			Seriman	
3.8	Value Checking	High level of return and parameter value checking			
4.7	Error Handling	Evidence of a systematic error handling mechanism			
3.0	Error Logging	Evidence of an error / fault logging mechanism			
3.8	Instrumentation	Evidence of run-time statistics and timing instrumentation			
2.9	Health Monitoring	Effective use of watchdog and system health monitoring			
3.2	SECURITY /				
3.5	Defensiveness	High level of defensive / secure coding practices		Kostial	
3.8	Validation	High level of validation of untrusted inputs / interfaces			
3.0	Libraries	Low level of use of potentially unsafe library calls			
2.3	Warnings	Evidence that code complies with no (or few) warnings			
3.9	Traceability	Effective use of executable and parameter file signatures / anti-tamper measures			
2.9	Density	Low density of secure coding issues as per SEI CERT Secure Coding Rules		Švoboda	
4.0	Priority	Low number of high-priority secure coding issues as per SEI CERT Secure Coding Rules			
3.4	TESTABILITY /			Seriman	
2.5	Unit	Evidence of unit testing, especially automated (test cases, test harness, etc)			
4.0	Regression	Evidence of regression testing, especially automated			
3.9	Complexity	Low valued cyclomatic complexity and path count			
3.17	TOTAL RISK				

- The Risk Results tab shows all risk score roll-ups and an overall risk score for the codebase.
 - The overall risk score derives from the weighted scores across all the analyst tabs.
- The Change field can be used to enter a $\pm$ delta from a prior CREW analysis when analyzing multiple code drops.
- The Analyst field can be used to parcel out & record who on the analysis team performed the analysis for each particular aspect or focus.

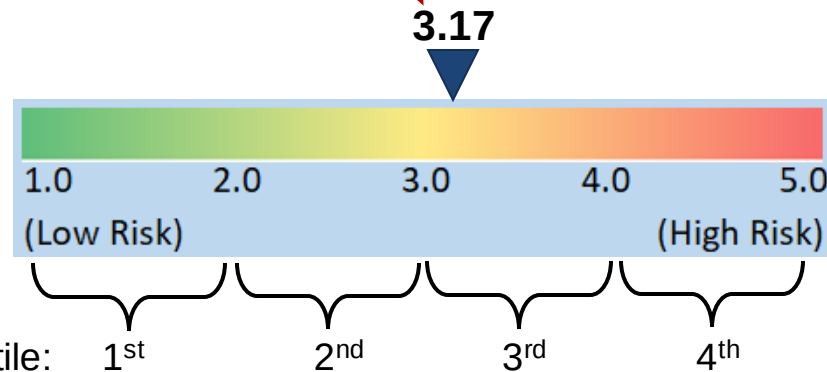
This score summary view is central in all SEI CREW code quality outbriefs.

Interpreting a CREW Risk Assessment



- As we've seen, CREW utilizes both scores and colors to indicate code quality risks.
 - But how might we notionally interpret a total code risk score and color for a particular critical system?

Quartile	Code Quality	Implication
1	Production	Ready
2	Solid	Improve
3	Substandard	Refactor
4	Inferior	Redesign



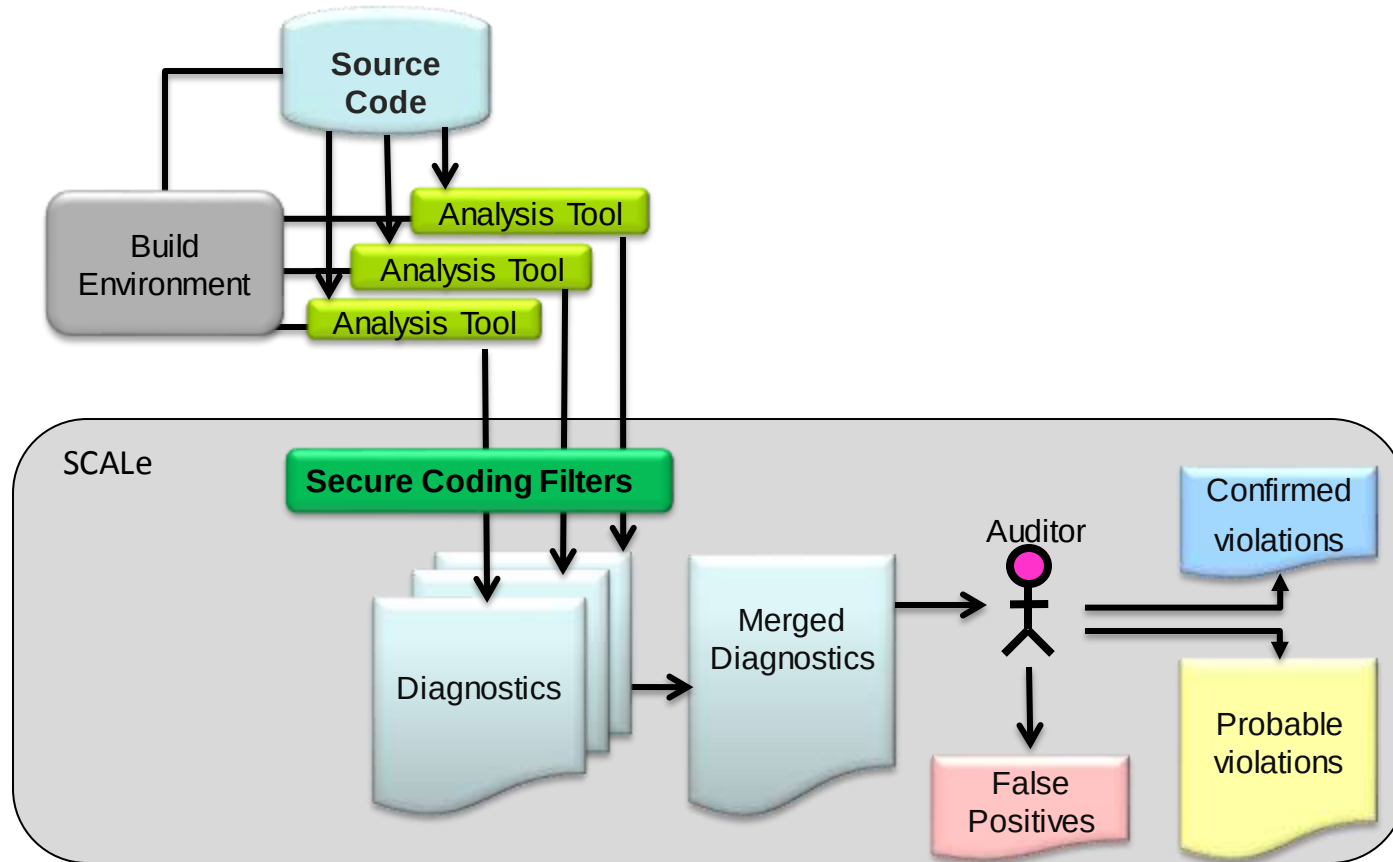
CREW scores may harmonize critical system code quality with implications for deployment.

Assessing Code Quality and Security

Overview of SCALe for Code Security Analysis

David Svoboda

SCALE Secure Coding Conformance Process



How to Audit Thousands of Static-Analysis Alerts?

- If you find 3 true positives that violate the same coding guideline:
 - Mark them **True** AND
 - Mark all other alerts of that guideline as **Suspicious**.
- If you find 15 false positives *at random*¹ from the same checker:
 - Mark them **False** AND
 - Mark all other alerts of that checker as **Ignored**.

¹ This means 95% confidence that the false positive rate (FPR) of the checker is over 80%.

Selected SCALe Assessments – Historical Data

Codebase	Customer	Lang	ksLOC	Rules	Diags	True	Suspect	Diag /KsLOC
A	Gov1	C++	38.8	12	1,071	52	1,019	27.6
B	Gov1	C	87.4	28	17,543	86	17,457	200.7
C	Gov2	C	9,585	18	289	159	130	0.03
D	Gov3	Java	4.27	18	345	117	228	80.8
E	Gov2	Java	61.2	33	538	288	250	8.8
F	Gov2	Java	17.6	21	414	341	73	23.5
G	Gov4	Java	653	29	8,526	64	8,462	13.1
H	Gov5	Java	1.51	8	53	53	0	35.1
I	Mil1	Java	403	27	3114	723	2,391	7.7

Assessing Code Quality and Security

Overview of SCALe for Code Security Analysis

The CERT Secure Coding Rules for C/C++

David Svoboda

CERT Secure Coding Standards



Collected wisdom from thousands of contributors on community wiki since Spring 2006

SEI CERT C Coding Standard

- Free PDF download:

<http://cert.org/secure-coding/products-services/secure-coding-download.cfm>

- Basis for ISO TS 17961 C Secure Coding Rules



SEI CERT C++ Coding Standard

- Free PDF download (Released March 2017):

<http://cert.org/secure-coding/products-services/secure-coding-cpp-download-2016.cfm>

CERT Oracle Secure Coding Standard for Java

“Current” guidelines available on CERT Secure Coding wiki

- <https://www.securecoding.cert.org>

Rules and Recommendations

Rules must meet the following criteria:

- Violation is likely to result in a defect that may adversely affect the safety, reliability, or security of a system.
- Does not rely on source code annotations or assumptions.
- Conformance can be determined through automated analysis (either static or dynamic), formal methods, or manual inspection techniques.

Recommendations are suggestions for improving code quality. Guidelines are defined to be recommendations when all of the following conditions are met:

- Application is likely to improve the safety, reliability, or security of software systems.
- One or more of the requirements necessary for a rule cannot be met.

There are 103 rules and 169 recommendations in the CERT C Coding Standard.

CERT Rule Components 1

Pages / ... / Rec. 01. Declarations and Initialization (DCL) Edit Watch Share ...

DCL22-CPP. Functions declared with `[[noreturn]]` must return void

Created by Aaron Ballman, last modified on Aug 24, 2016

As described in [MSC55-CPP: Do not return from a function declared `\[\[noreturn\]\]`](#), functions declared with the `[[noreturn]]` attribute must not return on any code path. If a function declared with the `[[noreturn]]` attribute has a non-void return value, it implies that the function returns a value to the caller even though it would result in [undefined behavior](#). Therefore, functions declared with `[[noreturn]]` must also be declared as returning void.

Noncompliant Code Example

In this noncompliant code example, the function declared `[[noreturn]]` is declared to return an `int`:

```
#include <cstdlib>

[[noreturn]] int f() {
    std::exit(0);
    return 0;
}
```

This example does not violate [MSC55-CPP: Do not return from a function declared `\[\[noreturn\]\]`](#) because `std::exit()` is declared `[[noreturn]]`, so the `return 0;` statement can never be executed.

CERT Rule Components 2

Noncompliant Code Example

In this noncompliant code example, the function declared with `[[noreturn]]` claims to return an `int`:

```
#include <stdlib.h>

[[noreturn]] int f() {
    std::exit(0);
    return 0;
}
```

Noncompliant Code

Don't try this at home!

This example does not violate [MSC55-CPP: Do not return from a function declared `\[\[noreturn\]\]`](#) because `std::exit()` is declared `[[noreturn]]`, so the `return 0;` statement can never be executed.

Compliant Solution

Because the function is declared `[[noreturn]]`, and no code paths in the function allow for a return in order to comply with [MSC55-CPP: Do not return from a function declared `\[\[noreturn\]\]`](#), the compliant solution declares the function as returning `void` and elides the explicit return statement:

```
#include <stdlib.h>

[[noreturn]] void f() {
    std::exit(0);
}
```

Compliant Code

Fixes noncompliant code.

CERT Rule Components 3

Automated Detection

Tool	Version	Checker	Description
Clang	3.9	-Winvalid-noreturn	

Related Vulnerabilities

Search for [vulnerabilities](#) resulting from the violation of this rule on the [CERT website](#).

Related Guidelines

SEI CERT C++ Coding Standard	MSC54-CPP . Value-returning functions must return a value from all exit paths MSC55-CPP . Do not return from a function declared [[noreturn]]
--	--

Bibliography

[ISO/IEC 14882-2014]	Subclause 7.6.3, "Noreturn Attribute"
--------------------------------------	---------------------------------------

CERT Rule Components 4

Risk Assessment

A function declared with a non-void return type and declared with the `[[noreturn]]` attribute is confusing to consumers of the function because the two declarations are conflicting. In turn, it can result in misuse of the API by the consumer or can indicate an implementation bug by the producer.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL22-CPP	Low	Unlikely	Low	P3	L3

Automated Detection

Tool	Version	Checker	Description
Clang	3.9	-Winvalid-noreturn	

Assessing Code Quality and Security

Overview of SCALe for Code Security Analysis

The CERT Secure Coding Rules for C/C++
Rule Priority Levels

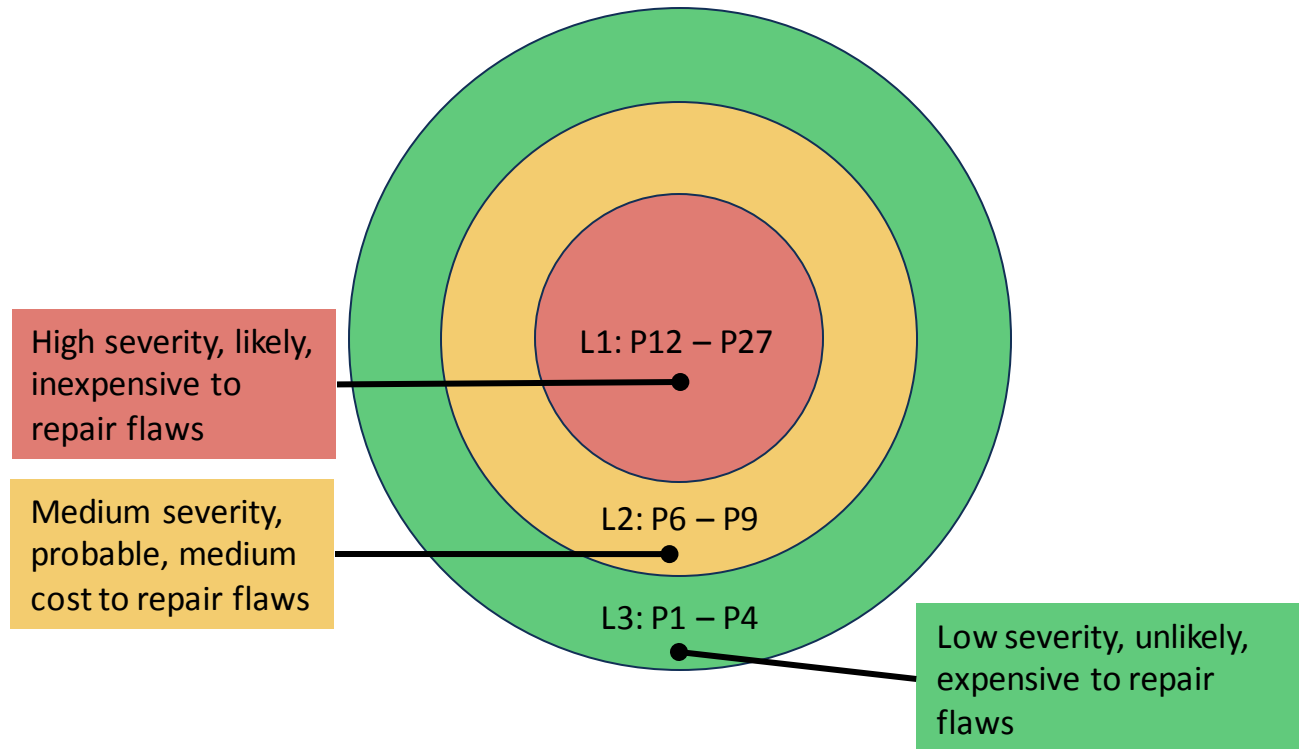
David Svoboda

Risk Assessment of Secure Coding Rule Violations

Risk assessment is performed using failure mode, effects, and criticality analysis.

Severity —How serious are the consequences of the rule being ignored?	Value	Meaning	Examples of Vulnerability	
	1	low	denial-of-service attack, abnormal termination	
	2	medium	data integrity violation, unintentional information disclosure	
Likelihood —How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?	3	high	run arbitrary code	
	Value	Meaning		
	1	unlikely		
	2	probable		
Cost —The cost of mitigating the vulnerability.	3	likely		
	Value	Meaning	Detection	Correction
	1	high	manual	manual
	2	medium	automatic	manual
	3	low	automatic	automatic

Rule Levels and Priorities



Assessing Code Quality and Security

Overview of SCALe for Code Security Analysis

Augmenting CREW Analyses with SCALe

Jay Marchetti

Augmenting Code Security Assessments: CREW + SCALE



Carnegie Mellon University
Software Engineering Institute

Code Risk Estimation Worksheet
CREW v3.0M
Software Engineering Institute ©2022 Carnegie Mellon University

Project: SystemX
Code Revision: 1.52
Language(s): C
Build System: Visual Studio 2019
Run-time OS: Windows 10
Run-time CPU: Intel Core i7
Analyst(s): Marchetti, Reimer, Kozlowski, Cohn, Sobotta
Review Date: 08/26/22

USAGE KEY
Aspect: The six (6) aspects of code quality being assessed for risk.
Focus: Sub-aspects that are rated via the nine (9) scoring tabs.
Best Practice: Focus best practice / positive code quality characteristic.
Change: Optional analyst's noted change in a rating from the assessment of a prior code drop (input [y/-delta]).
Analyst: Optional assigned analyst for an aspect (input [dev]).
Notes: Optional additional analyst's notes (input [note]).
Ratings: Rolled-up risks from the nine (9) details tabs that the mitigation statements are NOT true.

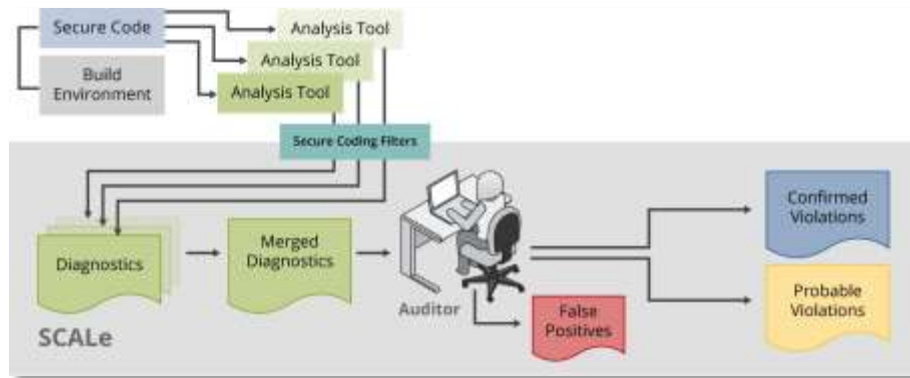
CODE RISK ESTIMATION WORKSHEET FOR: SystemX

ASPECT RISK	ASPECT / Focus	BEST PRACTICE	CHANGE	ANALYST	NOTES
2.4	ARCHITECTURE /			Marchetti	
2.1	Modularity	High level of code modularity and module cohesiveness			
2.4	Dependencies	Low level of dependencies, especially cyclic (DPA, dependency graphs, etc)			
2.7	Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)			
3.4	STANDARDS /			Kozlowski	
3.2	Style	High level of adherence to a chosen internal or published stylistic coding standard			
3.1	Clarity	High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)			
3.8	Language	High level of adherence to widely accepted language guidelines			
3.9	MAINTAINABILITY /			Cohn	
3.7	Quantity	High level of commenting excluding boilerplates			
3.2	Quality	High quality of commenting that clarifies the code's intent			
3.4	Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)		Marchetti	
2.7	Constants	High level of use of symbolic constants			
3.6	Files / folders	High level of source file and folder naming consistency and organization			
2.2	Readability	High level of overall code readability			
4.0	Conditionals	Low level of conditional compilation			

CREW

Risk Results Input System Architecture Standards Maintainability/Comments Maintainability/Clarity Maintainability/Readability

- CREW succinctly provides a broad-spectrum code security analysis by relying on probabilistic true-positives.
 - This is often all that is required for a given code assessment.
- SCALE delves deeper to provide analyst-verified true-positives.
 - When warranted, SEI can combine CREW and SCALE to provide a fuller code security assessment.



A SCALE audit can be used to corroborate a CREW code security assessment.

Assessing Code Quality and Security

Briefing CREW Analysis Results

Jay Marchetti

Briefing CREW Analysis Results



- SEI uses CREW screenshots and scores, tool outputs, analyst notes, and source code examples to communicate the code assessment results to stakeholders.

FOCUS	W	#	I	Measurement / Question	A	Finding	D
UNIT	2.6	0.350	1	Does the supplied code base include the unit test code?	Yes	0	0
			2	Do unit tests utilize a test harness with mocked dependencies and / or stubbed methods?	Somewhat	0	0
			3	Do unit tests appear designed to test edge cases, i.e. go beyond happy-path testing?	Some	0	0
			4	Do unit tests utilize assertions liberally to indicate test success or failure?	Some	0	0
			5	Does the coverage of the unit tests appear to be comprehensive?	Many	0	0
			6	Does the code base utilize global variables?	Minimal	0	0
			7	Does the test code allow easy documentation of application or library code?	No	0	0
			8	Do tests utilize both positive and negative test cases, i.e. fault injection?	Somewhat	0	0
			9	Are the unit tests generally easy to understand and verifiable?	No	0	0
REGRESSION	4.0	0.300	10	Is there evidence in either code or comments that the code is regression tested?	Some	0	0
			11	Is there evidence of test driver usage or other automated regression testing?	No	0	0
			12	Is there evidence of functional testing, i.e. testing slices of functionality within the whole system?	No	0	0
COMPLEXITY	3.9	0.350	13	Maximum cyclomatic complexity (CC1) for any function within the code base	• 100	0	0
			14	Total number of functions or methods within the code base	• 5000	0	0
			15	Percentage of functions or methods within the code with cyclomatic complexity (CC1) <= 10	80.0%	0	0
			16	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 10 and <= 20	8.0%	0	0
			17	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 20 and <= 30	6.0%	0	0
			18	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 30 and <= 40	4.0%	0	0
			19	Percentage of functions or methods within the code with cyclomatic complexity (CC1) > 40 and <= 50	1.8%	0	0
			20	Number of functions within the code with cyclomatic complexity (CC1) > 50	12	0	0
			21	Average file cyclomatic complexity (CC1)	• 3.33	0	0
			22	Average file strict cyclomatic complexity (CC2)	4.46	0	0
			23	Percentage of functions or methods within the code with strict cyclomatic complexity (CC2) > 10	25.0%	0	0
			24	Average file modified cyclomatic complexity (CC3)	3.00	0	0
			25	Percentage of functions or methods within the code with Hersteller PATH exceeding 80	3.7%	0	0
RISK	3.44						

Score colors, including those for individual analyst questions, help to create a briefing.

Assessing Code Quality and Security

Briefing CREW Analysis Results

Organization of a Typical CREW Briefing

Jay Marchetti

Organization of a Typical CREW Briefing

SEI's standard format for CREW Code Assessment outbriefs follow this basic organization:

- Bottom Line Up Front (BLUF) slide with 2 columns, one for characteristics to Maintain (left column) and one for issues to Improve (right column).
- Slide showing the 6 Aspects and 32 Focuses CREW uses to assess structural code quality,
- Overview slide showing the full Risk Results and highlighting a few top positives & negatives.
- System slide showing the System tab and highlighting a few top system observations.
- Then a series of slides consisting of 2 slides for each Analyst (scored) tab:
 - The first showing a Risk Results tab close-up of the scored tab's Aspect & Focus scores with one bullet (green bullet if ≤ 3.0 , red if > 3.0) per focus detailing a main finding.
 - The second showing a few tool outputs or code samples as finding exemplars.
- Summary slide which revisits the content of the BLUF slide.

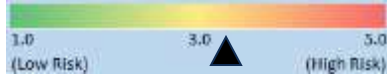
CREW outbriefs usually have about 24 content slides and last 90-120 minutes with Q&A.

SystemX 1.52 Code Risk Estimation

CREW Total Risk Score = **3.17**

MAINTAIN	IMPROVE
Good module & code layout consistency and file & folder organization	Reduce the number of conditional compilation flags in the code and / or make them run-time traceable
Uniform adherence to your stylistic SystemX C Coding Standard	Adopt and adhere to a critical coding standard such as MISRA or JSF++ and enforce it via static analysis
Good machine independence / portability	Eliminate dynamic memory allocation after boot-time initialization
Solid unit testing though there is no evidence of automated regression testing	Eliminate remaining duplicated source files and copy / pasted code sections
Some use of run-time monitoring	Implement traceability of data & executable files with anti-tamper verification at boot-time
Excellent accuracy, level of, and clarity of code commenting	Reduce the number of cyclic and bi-directional dependencies
Lower density of secure coding issues	Reduce the depth of control structure & call nesting

Overview



❑ SystemX code quality can be improved.

- The CREW total risk score of **3.17** indicates that there are code quality challenges.
- Architecture is a stand out characteristic – implying a solid basis for quality enhancements.
- SEI recommends improvements targeting fault tolerance, security, and testability to strengthen SystemX code quality.
- Incorporating elements of a critical system coding standard and language best practices would benefit overall SystemX quality.

CODE RISK ESTIMATION WORKSHEET FOR: SystemX		
ASPECT RISK	ASPECT / Focus	BEST PRACTICE
2.4	ARCHITECTURE /	
	2.1 Modularity	High level of code modularity and module cohesiveness
	2.4 Dependency	Low level of dependencies, especially cyclic (DSM, dependency graphs, etc)
	2.7 Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)
3.4	STANDARDS /	
	2.5 Stylistic	High level of adherence to a chosen internal or published stylistic coding standard
	3.1 Criticality	High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)
	3.9 Language	High level of adherence to widely accepted language guidelines
2.8	MAINTAINABILITY /	
Comments	1.7 Quantity	High level of commenting excluding boilerplates
	2.2 Quality	High quality of commenting that clarifies the code's intent
Clarity	1.4 Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)
	2.7 Constants	High level of use of symbolic constants
	1.6 Files / Folders	High level of source file and folder naming consistency and organization
	2.3 Readability	High level of overall code readability
Reusability	4.0 Conditionals	Low level of conditional compilation
	2.5 Independence	High level of machine, OS, database, and compiler independence
	3.4 Utilization	Low level of commented-out or dead code, and unused variables
	4.0 Complication	Low level of complicated implementation constructs
	2.9 Duplication	Low level of duplicated (copy / pasted) code
	FAULT TOLERANCE /	
	3.8 Value Checking	High level of return and parameter value checking
	4.7 Error Handling	Existence of a systematic error handling mechanism
	3.0 Error Logging	Existence of an error / fault logging mechanism
	3.8 Instrumentation	Existence of run-time statistics and timing instrumentation
	2.9 Health Monitoring	Effective use of watchdog and system health monitoring
3.2	SECURITY /	
General	3.6 Defensiveness	High level of defensive / secure coding practices
	3.8 Validation	High level of validation of untrusted inputs / interfaces
	3.0 Libraries	Low level of use of potentially unsafe library calls
	2.3 Warnings	Evidence that code compiles with no (or few) warnings
	3.9 Traceability	Effective use of executable and parameter file signatures / anti-tamper measures
CERT	2.9 Density	Low density of secure coding issues as per SEI CERT Secure Coding Rules
	4.0 Priority	Low number of high-priority secure coding issues as per SEI CERT Secure Coding Rules
3.4	TESTABILITY /	
	2.6 Unit	Evidence of unit testing, especially automated (test cases, test harness, etc)
	4.0 Regression	Evidence of regression testing, especially automated
	3.9 Complexity	Low valued cyclomatic complexity and path count
	3.17	TOTAL RISK

Assessing Code Quality and Security

Recommendations for Acing a CREW Analysis

Jay Marchetti

Recommendations for Acing a CREW Analysis

SEI offers these recommendations for improving mission-critical code quality:

#	ITEM	DESCRIPTION
1	Architecture	Plan the software architecture in advance, utilizing a known and appropriate architectural pattern and ensure that the code remains true to it throughout the SDLC.
2	Standards - Style	Adopt a written stylistic coding standard to be used consistently by all developers.
3	Standards - Criticality	Adopt (or tailor) a criticality coding standard to be used by all developers and ensure they are fully trained in it, as well as accepted best practices for the chosen language.
4	Static Analysis (SA)	Ensure all developers and the SQA team run at least one SA tool against all code they are responsible for and mitigate any high-priority true-positive diagnostics identified.
5	Metrics	Ensure all developers and the SQA team periodically review volume (PSLOC) and code complexity (file max function CC) metrics for all code they are responsible for.
6	Unit Testing	Adopt a UT framework and ensure all developers use it to verify the code they are responsible for prior to VCS check-in, and that all UT code and test cases are also maintained in the VCS.
7	DevSecOps	Adopt a DevSecOps continuous integration build & test pipeline that includes SA and regression testing.
8	CREW	Ensure all developers and the SQA team have reviewed the CREW Analyst questions.

Following these recommendations will lead to superior code quality.

Assessing Code Quality and Security

Potential Future CREW Improvements

Jay Marchetti

Potential Future CREW Improvements

SEI's Code Risk Estimation Worksheet (CREW) is a work in-progress.

Our team continues seeking ways to improve CREW and the code quality risk assessment processes around it, including:

- Adding new custom analytics to help analysts provide accurate CREW answers more quickly.
- Mapping additional SA tool outputs, standards, and metrics into CREW to strengthen the analysis.
- Combining CREW with a DevSecOps continuous integration build & test pipeline.

Code Risk Estimation Worksheet
CREW v5.20
Software Engineering Institute ©2022 Carnegie Mellon University

Project: SystemX
Code Revision: 1.10
Language(s): C
Run-time OS: Windows 9.8
Run-time CPU: Unknown
Analyst(s): Marchetti, Reimer, Kistal, Cohn, Subbala
Review Date: 08/16/22

USAGE KEY
The six (6) aspects of code quality being assessed for risk:
Aspect: Sub-aspects that are rated via the nine (9) scoring table.
Best Practice: Focus best practice / positive code quality characteristics.
Change: Optional analyst's noted change in a rating from the assessment of a prior code drop (input [4]-Delta[5]).
Analyst: Optional assigned analyst for an aspect (input [Delta]).
Notes: Optional additional analyst's notes (input [Delta]).
Rating: Rolled-up risks from the nine (9) details table that the mitigation statements are NOT true.
5.0 (Low Risk) 3.0 (High Risk)

CODE RISK ESTIMATION WORKSHEET FOR: SystemX

ASPECT	RISK	ASPECT / Item	BEST PRACTICE	CHANGE	ANALYST	NOTES
2.4	Architecture	2.1 Modularity	High level of code modularity and module cohesiveness		Marchetti	
		2.4 Dependency	Low level of dependencies, especially cyclic (DAM, dependency graphs, etc)			
		2.7 Portability	High level of adherence to a known architectural pattern (e.g., client-server, etc)			
3.4	Standards	2.3 Style	High level of adherence to a chosen internal or published standards coding standard		Kistal	
		3.1 Quality	High level of adherence to critical system coding guidelines (MISRA, CERT, etc)			
		3.2 Language	High level of adherence to widely accepted language guidelines			
3.8	Maintainability	3.7 Quantity	High level of commenting/excluding boilerplates		Cohn	
		3.2 Quality	High quality of commenting that clarifies the code's intent			
		3.4 Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)		Marchetti	
		2.7 Consistency	High level of use of symbolic constants			
		3.2 Flow / Editors	High level of source file and folder naming consistency and organization			
		3.2 Readability	High level of overall code readability			
		4.0 Conditionals	Low level of conditional compilation			

➤ Risk Results Inputs System Architecture Standards MaintainabilityComments MaintainabilityClarity MaintainabilityTestability

Have a CREW improvement idea? Please share it with our team...

Assessing Code Quality and Security

Summary

Assessing Code Quality and Security

Mike Riley

Summary

SEI developed CREW and SCALE to help us improve code assessments for our customers.

Our further intent, expressed through this course, is that these tools can also help your organization to produce higher quality code.

Code Risk Estimation Worksheet
CREW v5-20
Software Engineering Institute - ©2022 Carnegie Mellon University

Project: SystemX
Code Revision: 1.52
Language(s): C
Run-time OS: VxWorks 6.9
Run-time CPU: Unknown
Analyst(s): Marchetti, Reimer, Kostial, Cohn, Svalbode
Review Date: 08/16/22

USAGE KEY
Aspect: The six (6) aspects of code quality being assessed for risk.
Focus: Sub-aspects that are rated via the nine (9) scoring tabs.
Best Practice: Focus best practice / positive code quality characteristic.
Change: Optional analyst's noted change in a rating from the assessment of a prior code drop (input [+/- delta]).
Analyst: Optional assigned analyst for an aspect (input [text]).
Notes: Optional additional analyst's notes (input [text]).
Ratings: Rolled-up risks from the nine (9) details tabs that the mitigation statements are **NOT** true.
1.0 (Low Risk) 3.0 5.0 (High Risk)

CODE RISK ESTIMATION WORKSHEET FOR: SystemX

ASPECT RISK	ASPECT / Focus	BEST PRACTICE	CHANGE	ANALYST	NOTES
2.4	ARCHITECTURE /			Marchetti	
	2.1 Modularity	High level of code modularity and module cohesiveness			
	2.4 Dependency	Low level of dependencies, especially cyclic (DSM, dependency graphs, etc)			
	2.7 Patterns	High level of adherence to a known architectural pattern (layered, client-server, etc)			
3.4	STANDARDS /			Kostial	
	2.5 Stylistic	High level of adherence to a chosen internal or published stylistic coding standard			
	3.1 Criticality	High level of adherence to critical-system coding guidelines (MISRA, CERT, etc)			
	3.9 Language	High level of adherence to widely accepted language guidelines			
2.8	MAINTAINABILITY /				
Comments	1.7 Quantity	High level of commenting excluding boilerplates		Cohn	
	2.2 Quality	High quality of commenting that clarifies the code's intent			
Clarity	1.4 Consistency	High level of code stylistic consistency (naming, source layout, brace usage, etc)		Marchetti	
	2.7 Constants	High level of use of symbolic constants			
	1.6 Files / Folders	High level of source file and folder naming consistency and organization			
	3.2 Readability	High level of overall code readability			
	4.0 Conditionals	Low level of conditional compilation			

* Risk Results Inputs System Architecture Standards MaintainabilityComments MaintainabilityClarity MaintainabilityReusability

Thank you for your time and attention!

Contacts

Michael Riley

Senior Member of the Technical Staff
Initiative Lead / Resilient Critical Systems
Software Engineering Institute

Email: mariley@sei.cmu.edu

Phone: +1 412-268-6377

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Email: info@sei.cmu.edu

Phone: +1 412-268-5800

Jay Marchetti

Senior Member of the Technical Staff
System & Software Development and Analysis
Software Engineering Institute

Email: jaym@sei.cmu.edu

Phone: +1 412-268-8079

Questions & Answers

Backup Slides

Quality Attributes – Some Examples Defined

QA	Meaning / Relevance [2]
Usability	Ease with which the user can accomplish a desired task
Availability	Ready to perform whenever needed; encompasses reliability
Performance	Ability to meet system timing requirements
Testability	Ease of demonstrating the software's faults through testing
Modifiability	Minimized cost and risk of making changes to the software
Interoperability	Ease of exchanging information via interfaces with other systems
Safety	Avoidance of states that may damage or cause injury to external actors
Integrity	Assurance that information is accurate, complete, and valid, and has not been altered by unauthorized actions

Scoring Risk – Selections, Weights, & Inverted Risk

- ❑ Unlike metric or estimate questions, selection answers are merely (evenly) spaced values per the number of choices across the interval [0.0, 1.0]. So, they use a gain ($m = 1.0$) and offset ($b = 0$) to transform the X-axis before normalizing to (Low, Med, High).
 - **Example:** Selection inputs of [None, Few, Some, Many, All] would have values of [0.0, 0.25, 0.5, 0.75, 1.0] which are then applied directly to the (Lo, Med, High) member functions.
- ❑ Each question has a relative weight value in [0.0, 1.0] that multiplies the low, medium, high components of the input. This affects the input's degree of importance in the risk computation for that focus.
 - The resultant crisp risk output is divided by the average of all relative weights to maintain [0.0, 1.0] normal
- ❑ Answers are assumed to correlate positively with risk, i.e. higher input value = higher code risk.
 - That is, unless the question's InvertRisk field = 1, in which case it correlates inversely with code risk. This provides for flexibility in wording the analyst questions.

Scoring Risk – Computational Overview

FOR EACH INPUT:

- ❑ Scale the raw analyst value according to the abscissa gain and offset for that input to obtain x .
- ❑ Invert the scaled value if it is inversely proportional to code quality risk as indicated by the `InvertRisk` field, calculating $X = (1.0 - x)$.
- ❑ Normalize X to its 3 components Z_i using the Low, Med, High membership functions.
- ❑ Multiply each of the 3 components Z_i by the scalar relative weight (Weight) for that input.

THEN, FOR EACH FOCUS:

- ❑ Average all weighted member components to obtain the 3 average values for Low, Med, and High membership components ZAV_i .
- ❑ Obtain a crisp result by taking ZAV_i dot product with $[0.2, 0.5, 1.0]$ and divide the resulting scalar risk by the average of the relative weights for that focus to obtain a crisp risk in $[0.2, 1.0]$.
- ❑ Last, multiply the scalar crisp risk by 5.0 to cover the $[1.0, 5.0]$ risk interval.

SCALE Audit: Simple Code Example

```
int main(int argc, char **argv) {  
    int val;  
    int result;  
    if (argc != 2) {  
        printf("Usage: %s <integer>\n", argv[0]);  
        return 1;  
    }  
    read_integer(argv[1], &val);  
    result = val * val;  
    printf("%d\n", result);  
    return 0;  
}
```

Alert to Audit

```
int main(int argc, char **argv) {  
    int val;  
    int result;  
    if (argc != 2) {  
        printf("Usage: %s <integer>\n", argv[0]);  
        return 1;  
    }  
    read_integer(argv[1], &val);  
    → result = val * val;  
    printf("%d\n", result);  
}
```

Line	Message	Checker	Tool	Rule	Title
20	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

CERT Coding Rule EXP33-C

The screenshot shows a web browser window with the address bar displaying <https://www.securecoding.cert.org/confluence/display/c/EXP33-C.+Do+not+read+u...>. The page header includes the CERT logo and the text "Software Engineering Institute Carnegie Mellon University". The Confluence navigation bar shows "Spaces", "People", and a "Create" button. The main content area is titled "EXP33-C. Do not read uninitialized memory" and is categorized under "Rule 03. Expressions (EXP)". The page was created by Jeff Gennari and last modified by Will Snively 9 minutes ago. The text explains that local, automatic variables assume unexpected values if read before initialization, citing the C Standard, 6.7.9, paragraph 10, and ISO/IEC 9899:2011. It states that if an object with automatic storage duration is not initialized explicitly, its value is indeterminate. It also references undefined behavior 11 and notes that local, automatic variables stored on the program stack default to whatever values are currently stored in stack memory.

More Information

```
int main(int argc, char **argv) {
```

Additional messages for diagnostic 724		
Message	Line	Path
Using uninitialized value "val".	<u>20</u>	/scale_examples_code/EXP33-C/exp33- XXXXXXXXXX
Declaring variable "val" without initializer.	<u>11</u>	/scale_examples_code/EXP33-C/exp33- XXXXXXXXXX .c
Condition "argc != 2", taking false branch	<u>14</u>	/scale_examples_code/EXP33-C/exp33- XXXXXXXXXX

→ `result = val * val;`

`printf("%d\n", result);`

Line	Message	Checker	Tool	Rule	Title
<u>20</u>	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

More Information ₂

```
int main(int argc, char **argv) {  
    int val;  
    int result;  
    if (argc != 2) {  
        printf("Usage: %s <integer>\n", argv[0]);  
        return 1;  
    }  
    read_integer(argv[1], &val);  
    → result = val * val;  
    printf("%d\n", result);  
}
```

Line	Message	Checker	Tool	Rule	Title
20	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

The `read_integer()` function

```
int main() {
    void read_integer(char *s, int *val) {
        if (val == NULL) {
            return;
        }
        sscanf(s, "%d", val);
    }

    read_integer(argv[1], &val);
    result = val * val;
    printf("%d\n", result);
}
```

What if user does not provide a valid `int` in `argv[1]`?

`sscanf()` could fail and leave `val` uninitialized!

Line	Message	Checker	Tool	Rule	Title
20	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

Marking the Alert

```
int main(void) {  
    void read_integer(char *s, int *val) {  
        if (val == NULL) {  
            return;  
        }  
        sscanf(s, "%d", val);  
    }  
}
```

☐	ID	Flag	Verdict	Previous	Path	Line	Message
☐	624	[]	[Unknown]	Unknown	/scale_examples_code/EXP33-C/exp33- XXXXXXXXXX .c	26	Using uninitialized value "val". More
☐	724	[]	✓ [Unknown] [Ignored] [False] [Suspicious] [True]	Unknown	/scale_examples_code/EXP33-C/exp33- XXXXXXXXXX .c	20	Using uninitialized value "val". More
☐	924	[]	[Unknown]	Unknown	/scale_examples_code/INT31-C/int31- XXXXXXXXXX .c	26	Using uninitialized value "val". More
☐	1024	[]	[Unknown]	Unknown	/scale_examples_code/INT31-C/int31- XXXXXXXXXX .c	26	Using uninitialized value "val" when calling "malloc". More

A Second Code Example

```
int main(int argc, char **argv) {  
    int val;  
    int result;  
    if (argc != 2) {  
        printf("Usage: %s <integer>\n", argv[0]);  
        return 1;  
    }  
    read_integer(argv[1], &val);  
    → result = val * val;  
    printf("%d\n", result);  
}
```

Line	Message	Checker	Tool	Rule	Title
26	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

A changed `read_integer()` function

```
int main() {
    int rc;
    if (val == NULL) {
        return;
    }
    rc = sscanf(s, "%d", val);
    if (rc != 1) {
        printf("No integer found!\n");
        exit(1);
    }
    read_integer(s, &val);
    printf("%d\n", val);
}
```

→ `res`

Aborts if `sscanf()` did not initialize `val`

Line	Message	Checker	Tool	Rule	Title
26	Using uninitialized value "val". More	UNINIT	coverity	EXP33-C	Do not read uninitialized memory

Marking the Alert

```
int main() {  
    void read_integer(char *s, int *val) {  
        int rc;  
        if (val == NULL) {  
            return;  
        }  
        rc = sscanf(s, "%d", val);  
        if (rc != 1) {  
            // ...  
        }  
    }  
}
```

☐	ID	Flag	Verdict	Previous	Path	Line	Message
☐	624	[]	✓ [Unknown] [Ignored] [False] [Suspicious] [True]	Unknown	/scale_examples_code/EXP33-C/exp33- c	26	Using uninitialized value "val". More
☐	724	[]		Unknown	/scale_examples_code/EXP33-C/exp33- c	20	Using uninitialized value "val". More
☐	924	[]	[Unknown]	Unknown	/scale_examples_code/INT31-C/int31- c	26	Using uninitialized value "val". More
☐	1024	[]	[Unknown]	Unknown	/scale_examples_code/INT31-C/int31- c	26	Using uninitialized value "val" when calling "malloc". More

Bibliography

- [1] *Code Complete, Second Edition*, by Steve McConnell, Microsoft Press, 2004.
- [2] *Software Architecture in Practice, Third Edition*, by L. Bass, P. Clements, and R. Kazman, SEI Series in Software Engineering, Addison-Wesley, 2013.
- [3] *Better Embedded System Software*, by Philip Koopman, Drumnadrochit Press, 2010.
- [4] “Continuous Integration in DevOps,” *SEI Blog*, by C. Aaron Cois, January 2015, <https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html>
- [5] *A SLOC Counting Standard*, by Barry Boehm et al., Center for Systems and Software Engineering, USC, 2007, <http://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf>
- [6] “Measuring Architectural Complexity,” by Grady Booch, *IEEE Software*, July 2008. <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4548401>
- [7] *2IS55 Software Evolution (2)*, by A. Serebrenik, Eindhoven University of Technology, 2011.
- [8] *Software Engineering Concepts – Software Design Complexity*, University of Houston, http://sceweb.sce.uhcl.edu/helm/SWEN-SoftwareEngineeringConcepts/my_files/TableContents/Module-10/software_design_complexity.html

Bibliography

- [9] *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, T. McCabe et al., 1996. <http://www.mccabe.com/pdf/mccabe-nist235r.pdf>
- [10] *Introduction to Software Reliability: A State of the Art Review*, Reliability Analysis Center (RAC), 1996.
- [11] *Your Code as a Crime Scene*, by Adam Tornhill, Pragmatic Programmers, 2015.
- [12] *Bug Prediction at Google*, Google Engineering Tools, <http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html>
- [13] *Working Effectively with Legacy Code*, by Michael Feathers, Prentice Hall, 2005.
- [14] *Zen and the Art of Motorcycle Maintenance*, by Robert Pirsig, William Morrow, 1974.
- [15] *Exploring the Relationship Between Architecture Coupling and Software Vulnerabilities*, Working Paper 18-031, Harvard Business School, by Robert Lagerström, et al., 2018.
- [16] *Introduction to Software Testing*, 2nd ed. P. Ammann, J. Offutt, Cambridge University Press, ISBN 978-1107172012, 2017.
- [17] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990.
- [18] *Faults on its Sleeve - Amplifying Software Reliability Testing*, Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA), Cambridge, Massachusetts, USA, June 1993.