

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module 1: Introduction and Overview

Jay Marchetti

Copyright 2020 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.

Carnegie Mellon® and CERT® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM20-0726

Agenda:

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module	Duration	Topics	Discussing
1 	0:10 0:15 0:20	Introduction Overview Quality	Why Code Analysis? Why does the Program Office Care? Quality Attributes
2	0:10 1:00	Review Software Quality - 1	Coding Standards Static Analysis Metrics
3	0:10 1:00	Review Software Quality - 2	Architecture Common Code Quality Issues Dynamic Analysis
4	0:10 0:15 0:15 0:15	Review Software Security Software Acquisition Course Summary	Security, Standards, and Testing Acquisition Strategy Considerations: RFP's, Contract Execution, and Deployment Checklist for Acquiring High-Quality Code

Course Content – Two Tracks

Technology

- Show how code analysis leads to better software by shedding light on
 - architecture
 - code quality
 - security
- Introduce several important standards, metrics, tools, & techniques

Acquisition

- Recommend ways for acquirers to leverage the techniques presented here to obtain higher quality systems from contractors
- Describe what should be in your contract to ensure effective validation of code quality and security

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

Overview

Code Analysis

These are not three independent concepts...

- In fact they are highly inter-related.

Code analysis - from a technical viewpoint...

- Why is it important?

Code analysis can provide key insights into your software's:

- Architecture
 - Quality
 - Security
- 
- ✓ Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment and principles guiding its design and evolution.
 - ✓ Quality is the software's system-specific fitness for purpose, along with the underlying structural characteristics that make it so.
 - ✓ Security is the specific quality characteristic for thwarting attacks and is key for today's operational success.

Code Analysis: The Technical Context

What is “code analysis”?

- **Code analysis** is the automated analysis of software to assess the degree to which it embodies one or more of the 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
 - interoperability
 - safety
 - maintainability
 - security
 - testability
- While others may not be important for the system at hand, e.g.:
 - portability
 - scalability

Code Analysis: The Technical Context

Quality Attributes from code?

The 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 very difficult and expensive to add later.

- So, we apply **code analysis** early on and verify that the essential QAs have been designed into the system's "ground truth" artifact: the code.

TAKEAWAY

- Use code analysis early in the software development life cycle to verify essential QA's.
- Code analysis may sometimes be done "after the fact" to assess (or *audit*) whether code exhibits the desired (or *required*) QAs.

Quality Attributes Drill Down - Examples

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

Quality Attributes - Impacts to Software

Quality Attributes impact your system's architecture, code quality, and code security.

- Code analysis can be used to verify many high-priority QAs.

Impacting	Example Relevant QA
Architecture	Scalability, modularity, interoperability, safety
Code quality	Maintainability, testability, portability, safety
Security	Confidentiality, integrity

Question: The QAs below would most likely impact which: architecture, code quality, or security?

Modifiability → Code Quality?

Reliability → Architecture?

Availability → Security?

An argument could certainly be made that all 3, architecture, code quality, and security, are impacted by these QA's.

The Two Types of Code Analysis

There are two primary types of code analysis: **static** and **dynamic**.

Static analysis is the analysis of software while it is not executing. It is defect prevention.

Dynamic analysis is the analysis of software while it is executing. It is defect detection.

STATIC ANALYSIS (SA)	DYNAMIC ANALYSIS (DA)
Includes analyzing binaries, bytecode, or - most commonly (and our focus here) – the source code.	Includes runtime monitoring, fuzzing, and unit, integration, and system testing.

Use both ★ Example: Array bounds violation - Lint (SA tool) may find it via data flow analysis of source code, Purify (DA tool) may find it via tagged memory run-time instrumentation. Using both SA and DA increases the odds it is not missed!

TAKEAWAY

- Static and dynamic analysis are distinct but complimentary: use them both.
- V&V view: Static analysis is a part of verification; dynamic analysis is a part of validation.

Code Analysis: Why Does the Program Office Care? -1

1. So you know what you are buying

- Consider 3 cases involving software-intensive programs:

1. New Start with lots of Custom Code	2. New Start with lots of Reuse	3. Legacy Program with a Major Upgrade
“Bolting-on” quality later is prohibitively expensive – often effectively starting over.	Contractors bid high levels of reuse but seldom deliver those levels.	Baseline legacy code quality and performance before you start.
Evidence shows that built-in code quality lowers cost to your program over the long haul.	Validate off-the-shelf product architecture, code quality, and security assumptions.	

Code Analysis: Why Does the Program Office Care? -2

2. So you know what you are receiving

Code Quality Drives Costs throughout the System Lifecycle

Use code analysis to understand the code your contractor is planning to reuse.

Use code analysis to evaluate the quality of the code developed under your contract.

3. So you are ready for transition to operations

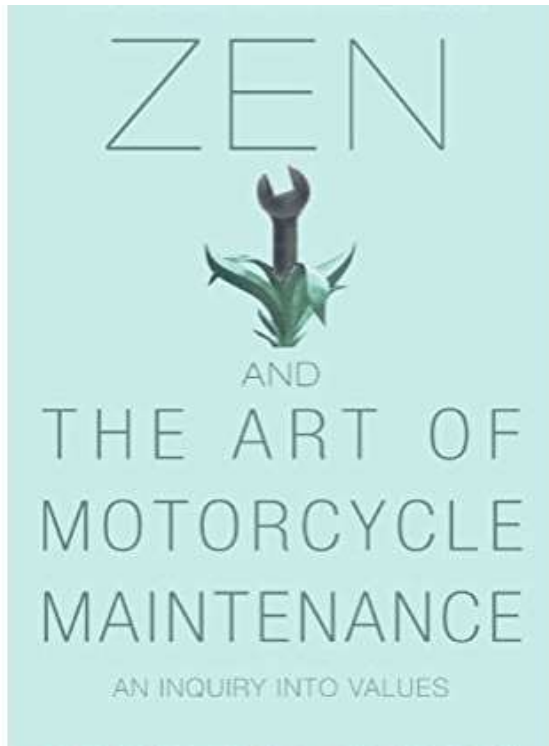
Code Analysis provides Audit Data

Code analysis provides evidence that you are actively managing software safety and / or security issues relevant to Program Protection Plan, Risk Management Framework, DO-178C, etc.

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis Quality

Quality – Hard to Define but Easy to Recognize



What is quality?

Quality is a central theme of the influential novel *Zen and the Art of Motorcycle Maintenance*, by Robert Pirsig.

“Quality ... you know what it is, yet you don’t know what it is. But that’s self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! ... But for all practical purposes, it does exist. ... Why else would people pay fortunes for some things and throw others in the trash pile?” [14]

Software Quality – Embodying the QAs

So again: What is software quality?

Software Quality is the degree to which software embodies the required QAs, thus indicating whether the it meets stakeholder needs [2].

External QAs (Execution)		Internal QAs (Evolution)	
Correctness	Usability	Maintainability	Flexibility
Efficiency	Reliability	Portability	Reusability
Integrity	Adaptability	Readability	Testability
Accuracy	Robustness	Understandability	Scalability
Performance	Availability	Interoperability	Deployability

- External QAs are evident to end users, internal QAs are evident mainly to developers.
- Internal QAs often affect external QAs: if it isn't **testable**, how do you know it's **reliable**?

Software Quality – Throughout the SDLC

Software quality cannot be bolted on – it must be built in

Software quality is a measure of how well software performs the function for which it was intended, including:

- Performing the intended job both initially and throughout the Software Development Life Cycle (SDLC)
- Encompassing all of the relevant QAs, both external and internal



While code analysis is the main subject of this course, we strongly assert this **Best Practice**:

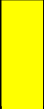


High-quality software starts with high-quality requirements and extends through architecture, code development, documentation, testing, deployment, and sustainment.

- High-quality software meets or exceeds all aspects of its intended purpose.
- You build quality into software throughout the Software Development Life Cycle.

Software Quality: A Common (but Wrong) Hierarchy of Priorities

Warning: The considerations that drive programs too often fall into 3 classes of priority...

Consideration		Priority	Overarching Associated Questions
Operationality		High	Does it work? Are all behavioral requirements met?
Cost and schedule		High	Will it be within budget and meet the targeted schedule?
Safety- and mission-criticality		Mid	Are all safety requirements and certifications met? Are all foreseeable unsafe or other critical failures mitigated?
Security		Mid	Are all IA requirements and certifications met? How difficult is it to exploit?
Non-behavioral requirements		Low	Does it incorporate the QAs, both external and internal, that have been deemed most important for the system?

TAKEAWAY

- Singular focus on the high- and mid-level priorities may gain short-term benefits, but at very high expense to longer-term costs.
- Strict adherence to these priorities often portends program failures downstream.

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

End of Module 1

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module 2: Software Quality₁

Jay Marchetti

Agenda:

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module	Duration	Topics	Discussing
1	0:10 0:15 0:20	Introduction Overview Quality	Why Code Analysis? Why does the Program Office Care? Quality Attributes
2 	0:10 1:00	Review Software Quality - 1	Coding Standards Static Analysis Metrics
3	0:10 1:00	Review Software Quality - 2	Architecture Common Code Quality Issues Dynamic Analysis
4	0:10 0:15 0:15 0:15	Review Software Security Software Acquisition Course Summary	Security, Standards, and Testing Acquisition Strategy Considerations: RFP's, Contract Execution, and Deployment Checklist for Acquiring High-Quality Code

Understanding Software Architecture, Quality, and Security Through Code Analysis

Review of Module 1

- This course has two tracks: the technical and acquisition aspects of code analysis.
- We defined code analysis in terms of the (tool automated) assessment of the code's quality attributes.
- We further introduced Quality Attributes (QAs) and noted that:
 - QAs can be viewed as *aggregates of code characteristics* that can be tested for at runtime or identified within the source code, i.e. they can be verified via code analysis.
 - QAs are very difficult and expensive to add later: start code analysis early in SDLC.
 - QAs may be classified as external (execution) and internal (evolution).
- We defined the two types of code analysis: static and dynamic.
- We looked at why the Program Office should care about code analysis:
 - So you know what you are buying.
 - So you know what you are receiving.
 - So you are ready for transition to operations.

Software Quality - 1

In this section, we'll cover these topics with respect to software quality:



- Coding standards
- Static analysis: verifying code *in vitro*
- Metrics

Coding Standards

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

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 System	Language restriction rules for secure, safety-critical, and mission-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 conventions are a best practice that should always be used.
- Additional standards for criticality and security are essential for some systems.

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

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 time and throughout the SDLC.

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.

TAKEAWAY

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

Critical System Coding Standards

A **critical system** is a software-intensive system that is safety-critical or mission-critical, or one for which security is a QA of primary importance.

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

- Additional effort, tools, and training for the system development → So: identify & address these needs **early** on in the project.
- The use of static analysis (SA) tools and / or compilers that support checking the code against criticality standards → a definite *best practice for all critical systems*.

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

Critical System Coding Standards

Critical system standards are rule sets that:

- Restrict the use of certain features of a programming language – e.g. malloc()
- Limit complexity and the amount of code within modules – e.g. Cyclomatic & SLOC
- Disallow dead code – i.e. unreachable, no effect, or commented out
- 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

TAKEAWAY

- Critical system coding standards essentially restrict the programmer's use of a language.

Mission-Critical and Safety-Critical Coding Standard Examples

MISRA: Motor Industry Software Reliability Association for C/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 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

Secure Coding Standards

- **SEI CERT** Secure Coding Standards for C, C++, Java, and Perl
- **CWE:** The MITRE Corporation's Common Weakness Enumeration for C/C++
- **OWASP:** Open Web Application Security Project

These coding standards are discussed later within the Security portion of the course.

Software Quality

In this section, we'll cover these topics with respect to software quality:

- Coding standards
- Static analysis: verifying code *in vitro*
- Metrics

Static Analysis: Verifying Code *in vitro*

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

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

- However, SA most often implies analysis of source code, which is our focus.

SA typically means automated source code analysis, i.e. using tools, though manual analysis – or peer review – is a best practice that may be considered a form of SA.

- This presentation targets source code analysis using SA tools.

- SA can be done on bytecode, binaries, or (most commonly) on source code.
- SA is generally automated via the use of one or more tools.

Static Analysis: Strengths

- Used early, SA helps developers eliminate code issues before they commit changes to their version control system.
- SA tools can analyze a large codebase in a short time.
- Many SA tools are free or inexpensive, allowing multiple tools to be applied cost-effectively.
- Many SA tools support security and critical coding standard checks.
- SA tools may be run automatically in a Continuous Integration (CI) pipeline.
- SA tools may detect subtle errors that can elude even experienced programmers.
- SA tools can identify issues that, while not errors, negatively impact code quality.



Static Analysis: Limitations



- SA tools often generate false positives, meaning the tool identifies issues that are not actually present. This can cause wasted time and loss of faith in the tool.
- SA tools will always have false negatives, meaning the tool will not find all code issues.
- SA tools may prove the existence of issues in code, but they do not prove the absence of issues. Misunderstanding this can lead to a false sense of security.
- SA tools cannot detect errors or vulnerabilities in the runtime environment.



Best Practice: As a developer, your compiler is your front-line SA tool – use it effectively! This means:

- Always build with warning levels maximized, e.g. for gcc use “-Wall”.
- Always ensure that the system builds with zero (0) warnings.
 - Any new warning emitted during a build will stand out so that it can be addressed!

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

TAKEAWAY

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

Static Analysis: A Sampling of Well-Known SA Tools

Tool	Vendor	Notes
Understand	Scientific Toolworks	Swiss-army knife SA tool, works on non-buildable source, supports many languages and MISRA
Coverity	Synopsys	Top-line SA tool, very expensive, many languages and standards
PC-lint	Gimpel Software	Venerable command-line Lint tool, C/C++ only, MISRA
PolySpace	MathWorks	Bug Finder and Code Prover, C/C++, many criticality standards
FindBugs	Open source	Well-known FOSS tool, Java only
CppCheck	Open source	Well-known FOSS tool, command line or GUI, C++ only
Lattix	Lattix	Special purpose tool for architectural views of C++ / Java code
Eclipse	Open source	IDE with many SA plugins available for C++ / Java code

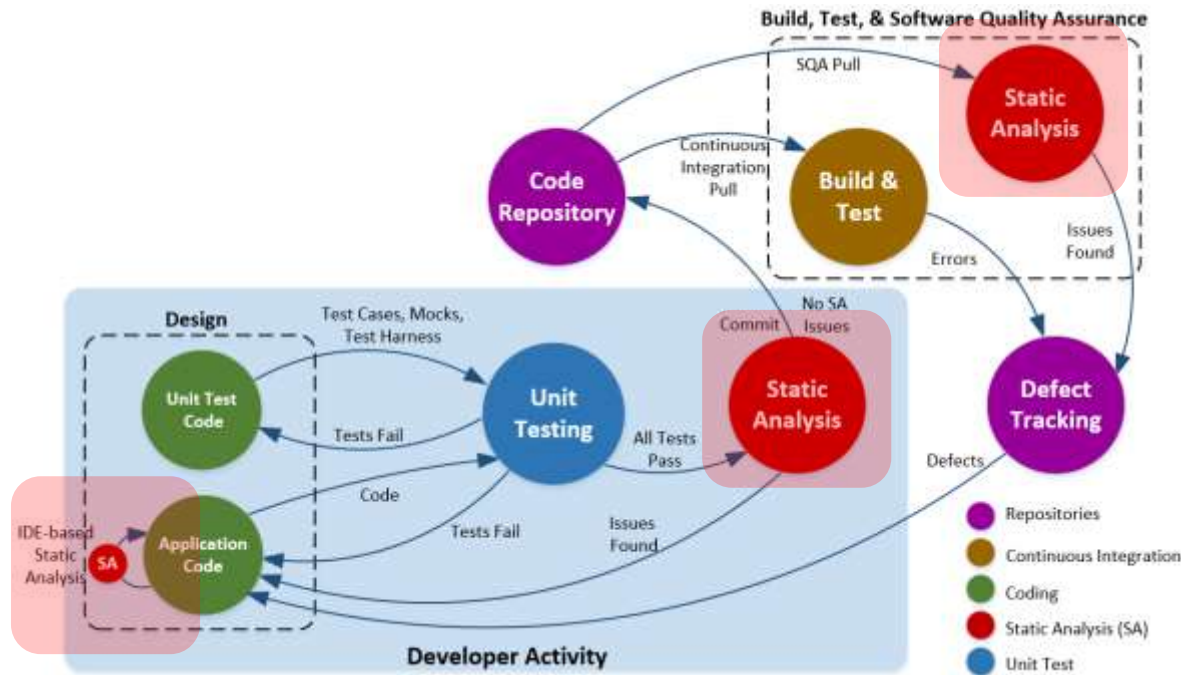
TAKEAWAY

- Good SA tools abound, though SEI does not endorse specific tools.

Static Analysis (SA)



Best Practice: Embedding SA within the Software Development workflow.

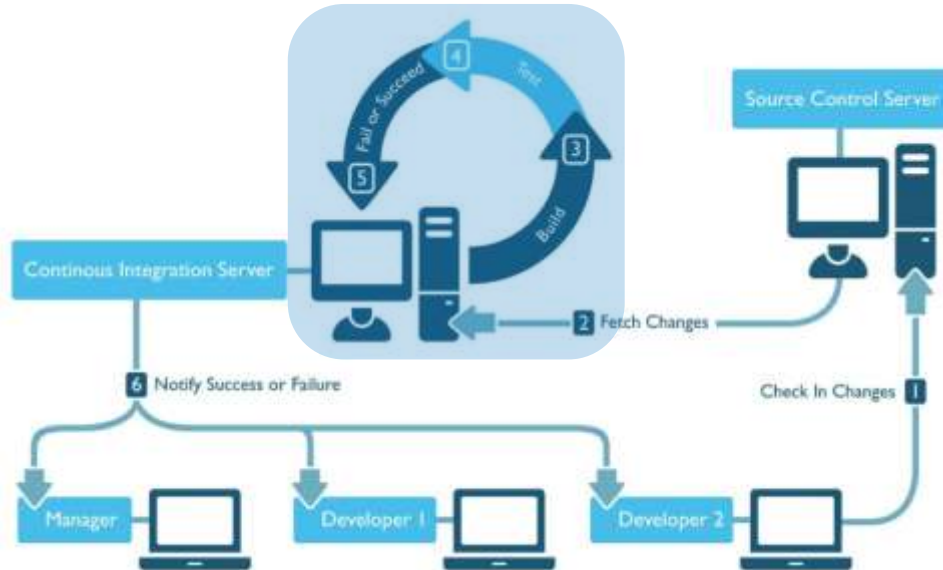


- Static analyses is done early and throughout the Software Development process.
- Ideally, three different SA tools are used here.

Static Analysis (SA)



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 suites.

Software Quality

In this section, we'll cover these topics with respect to software quality:

- Coding standards
- Static analysis: verifying code *in vitro*
- ▶ • Metrics

Metrics

Metrics are measures or counts of certain aspects of code that **provide insight into a variety of code QAs**.

Metric	Relevant QAs
Comment-to-Code Ratio	Descriptiveness, Modifiability
Maximum Nesting Depth	Understandability, Complexity
Average Dependency	Coupling, Reusability

Static Analysis (SA) tools typically support a wide variety of metrics. Here are just a few examples.

We'll cover **two** base metrics all developers and acquirers should know for their system:

Code Volume

Code Complexity

TAKEAWAY

- Experienced developers review metrics for their code as a tool to improve it.
- Acquirers should request regular metrics reports from contractors to put an SA deliverable up front.

Metrics: Code 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 code volume metric.

Two SLOC types with differing counting rules are most often cited: physical and logical SLOC:

Physical SLOC	Logical SLOC
Line count ignoring blank and comment lines	Logical statements (e.g., semicolon terminated)
Sensitive to code formatting and layout style	Insensitive to code formatting and layout style
Not language-specific	Language-specific counter program is required
Typically (2–3x) greater than logical SLOC	

TAKEAWAY

- When assessing SLOC, be sure you understand the counting rules being used.

Metrics: SLOC Misuse and Use

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



Key Idea: 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]

- Never use SLOC in the context of developer or contractor productivity.
- SLOC correlates with many key concerns for both new and existing code.

Metrics: SLOC Tools

SLOC metrics are generated by virtually all SA tools. However, many language-specific code counter tools are available:

Tool	Vendor	Notes
CLOC	Open source	Easy-to-use counter tool that auto-detects 150+ languages. http://cloc.sourceforge.net/
UCC	Open source	Unified Code Count, implements the code counting framework published by the SEI. Used by NASA and DoD. Also produces cyclomatic complexity and maintainability index metrics. http://csse.usc.edu/ucc_new/wordpress/
SLOCCount	Open Source	Easy-to-use counter tool that auto-detects 25+ languages. https://sourceforge.net/projects/sloccount/

Also, Linux provides (all) line counts via the command: `wc -l *.c *.h` for a C program.

TAKE
AWAY

- These free tools are a useful independent check of a contractor's reported SLOC.

Metrics: 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].



Key Idea: 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 one form of Technical Debt.

Metrics: Code Complexity

The more typical use of the term *complexity* with respect to software relates to the structural complexity of the source code itself, rather than the program's architecture.

There are two well-known code complexity metrics:

- **Cyclomatic Complexity (CC):** Proposed by McCabe (1976) to support basis path (aka structured) testing. This is, by far, the most used code complexity metric.
- **Halstead Complexity (HC):** From his “software science” work, halstead metrics are based on number of code operators and operands and go well beyond just complexity.

Coding standards for critical systems usually stipulate limits on CC for all functions. Why? Because Low CC correlates with **testability**.

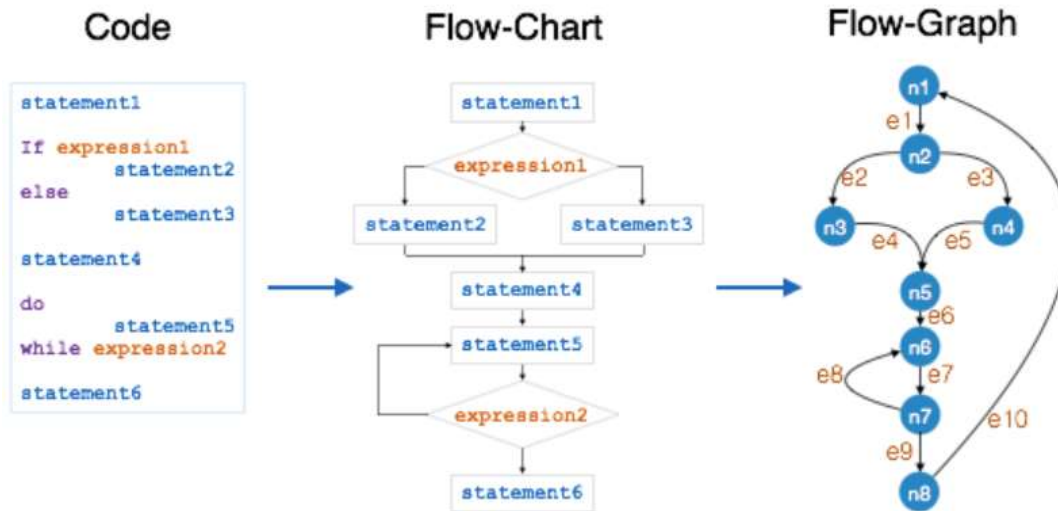
TAKEAWAY

- CC is the most common code complexity metric.
- Many standards for critical systems place strict upper limits on CC.

Metrics: Cyclomatic Complexity Measures Code Testability

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

Metrics: 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

Critical system coding standards limit the CC of all functions within an application.

- High CC is a **red flag** for code deserving close review, refactoring, and additional testing.

TAKEAWAY

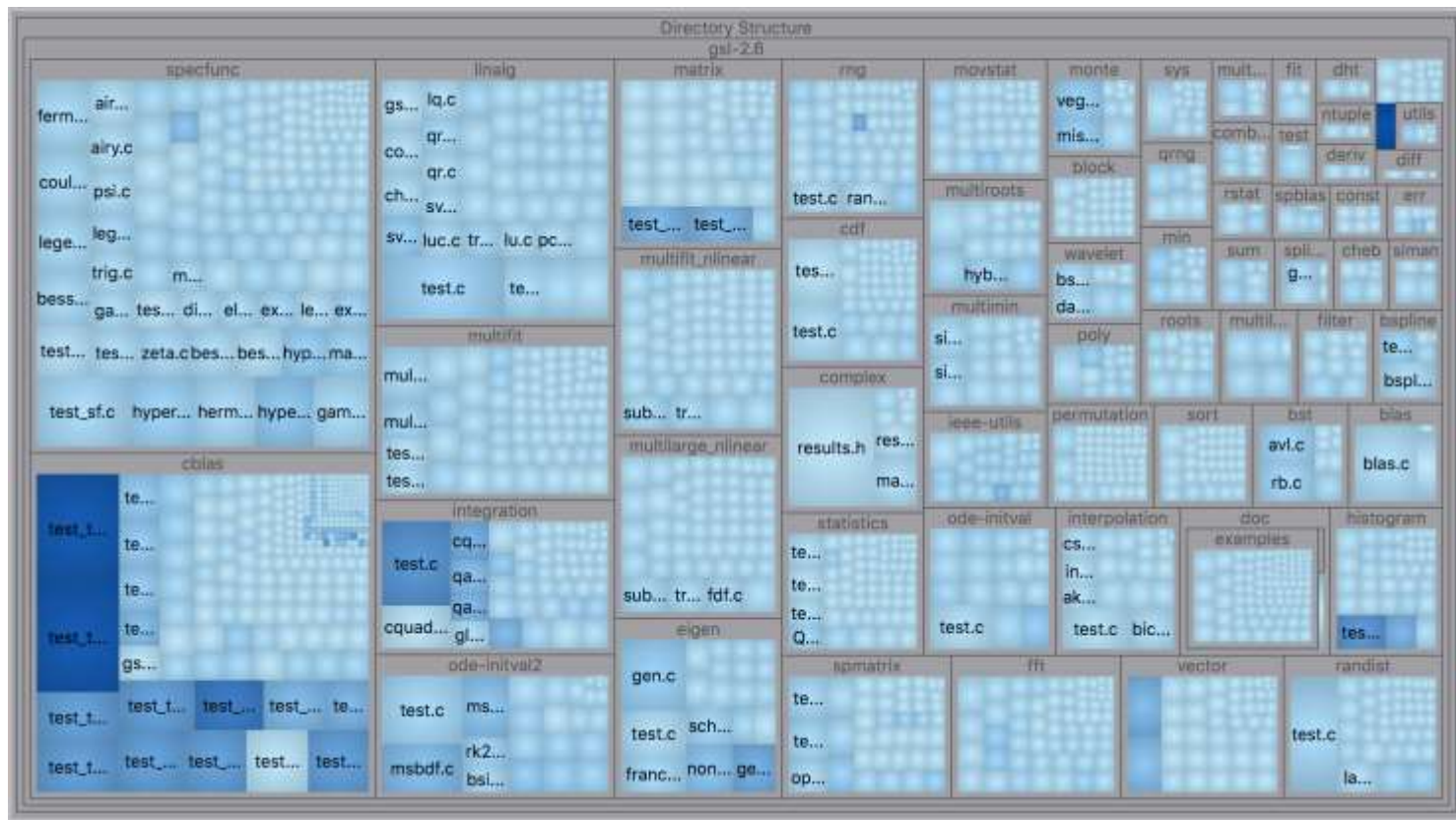
- High CC code is hard to test thoroughly. Unthoroughly tested code is less reliable.
- The CC metric is widely used for Quality Assurance and test planning purposes. [RIAC]

Cyclomatic Complexity – Treemap Visualization

The 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

Metrics: Complexity + Commits = Hot Spots

While “scholarly data on code complexity metrics is mixed because of the difficulty of performing controlled studies across different application areas, different programming team skill levels, different languages, and other confounding factors.” [Koopman]



Key Idea: Cyclomatic Complexity (CC) remains a useful **signal** for code that is difficult to understand, difficult to test, and more prone to defects.

However, defect prediction can be further enhanced by incorporating the commit history data from a version control system (VCS) such as Git.

- A code defect **hot spot** is the intersection of high-complexity code and high code-change frequency.
- Hot spots identify difficult code that has been worked on extensively. Research has shown such code to have a higher likelihood of latent defects [11].

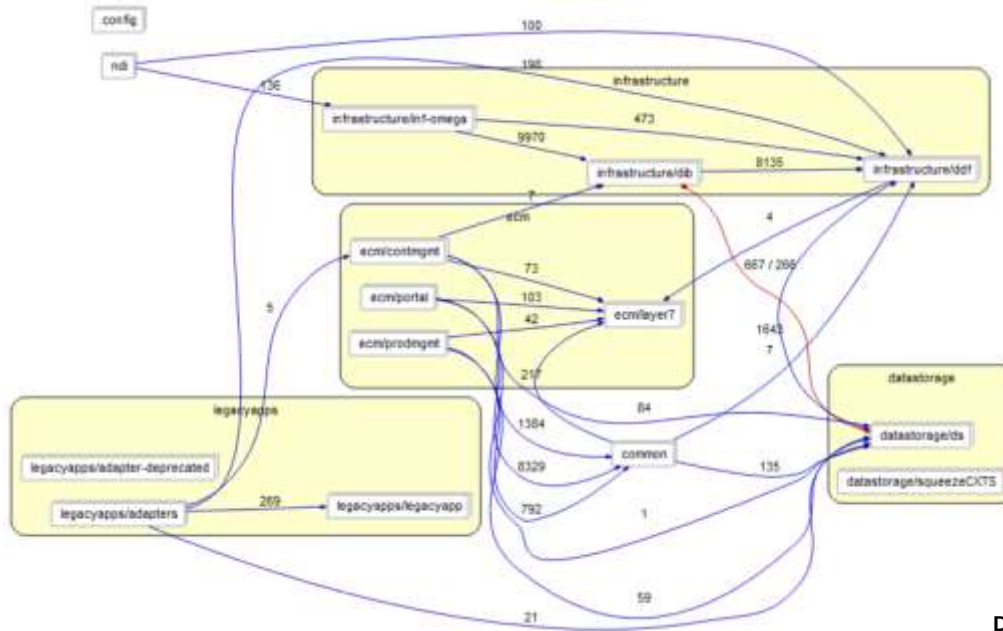
TAKE
AWAY

- Using code complexity metrics plus VCS history shows promise for bug prediction.

Metrics: Architectural Complexity – From Code Analysis?

Given a legacy system, where all we have is the codebase, is there any aspect of architectural complexity (AC) that is evident purely from SA tools?

- Yes, we can look at the number and types of interdependencies among modules or classes using dependency graphs.



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

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

End of Module 2

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module 3: Software Quality₂

Jay Marchetti

Agenda:

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module	Duration	Topics	Discussing
1	0:10 0:15 0:20	Introduction Overview Quality	Why Code Analysis? Why does the Program Office Care? Quality Attributes
2	0:10 1:00	Review Software Quality - 1	Coding Standards Static Analysis Metrics
3 	0:10 1:00	Review Software Quality - 2	Architecture Common Code Quality Issues Dynamic Analysis
4	0:10 0:15 0:15 0:15	Review Software Security Software Acquisition Course Summary	Security, Standards, and Testing Acquisition Strategy Considerations: RFP's, Contract Execution, and Deployment Checklist for Acquiring High-Quality Code

Understanding Software Architecture, Quality, and Security Through Code Analysis

Review of Module 2

- We discussed coding standards are a critical first step towards code quality. There are two (2) common types of coding standards:
 - Stylistic – Reduce likelihood of coding errors & enhance code maintainability.
 - Critical system – Restrict language features, limit complexity, etc. for safety- and mission-critical code for which they're essential, but will require extra tools & training.
- We introduced static analysis (SA): analysis of source code while it is not executing.
 - Best practice: Use SA tools early and throughout the SDLC: development through CI.
 - Your compiler is your first-line SA tool – ensure your system builds with zero warnings.
 - Utilize multiple dedicated SA tools from different manufacturers whenever possible.
- We defined code metrics and focused on two (2) basic metrics stakeholders must know:
 - Code volume (SLOC) correlates with many key concerns for both new and existing code bases, e.g. maintenance, migration, and new feature costs, etc.
 - Code complexity, i.e. cyclomatic complexity (CC), which measures code testability.

Software Quality - 2

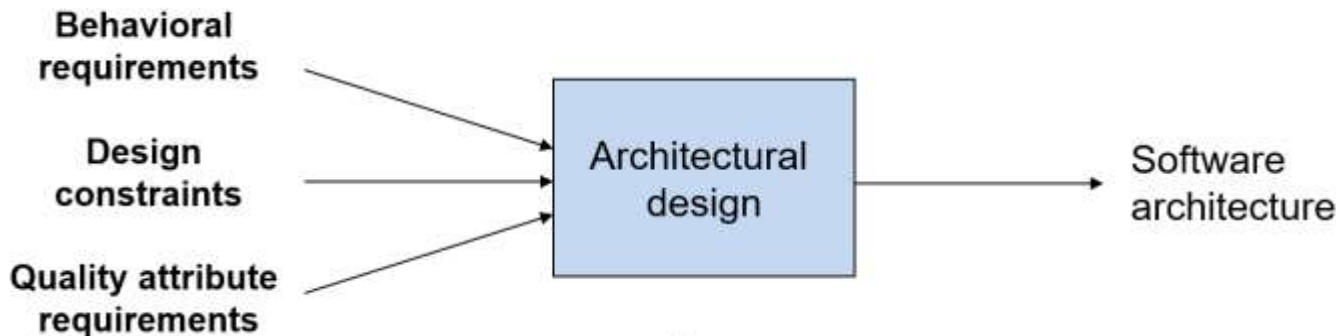
In this section, we'll cover these topics with respect to software quality:

- 
- Architecture
 - Code quality issues commonly detected using static analysis
 - Dynamic analysis: verifying code *in vivo*

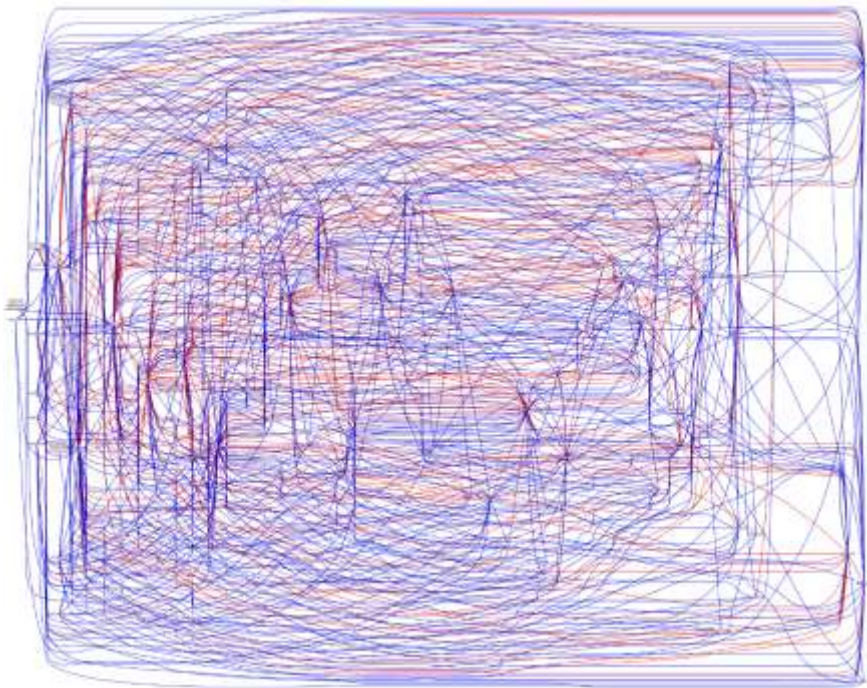
Architecture

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.



Architecture – From Code Analysis



Dependency Graphs

← For large or complex systems, dependency graphs quickly get out of hand.

Yet, given a code base we need *some* insight into its architecture to:

- Effectively change or maintain the system.
- Assess potential reuse of system components.

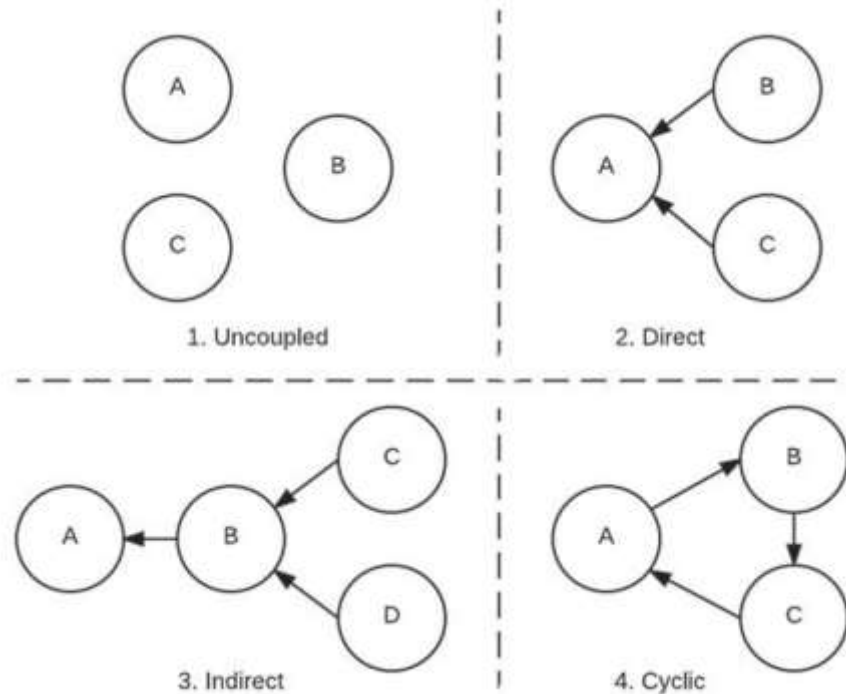
But reengineering architecture from code is a hard problem that's not fully automatable.

- TAKEAWAY**
- Extracting architecture from code analysis for large or complex systems is analyst-intensive.
 - SA tools help, but no current tool can automate the process.

Architecture – Dependency Types

Modules within a system may be dependent in several ways:

1. Uncoupled
2. Directly coupled
3. Indirectly coupled
4. Cyclic

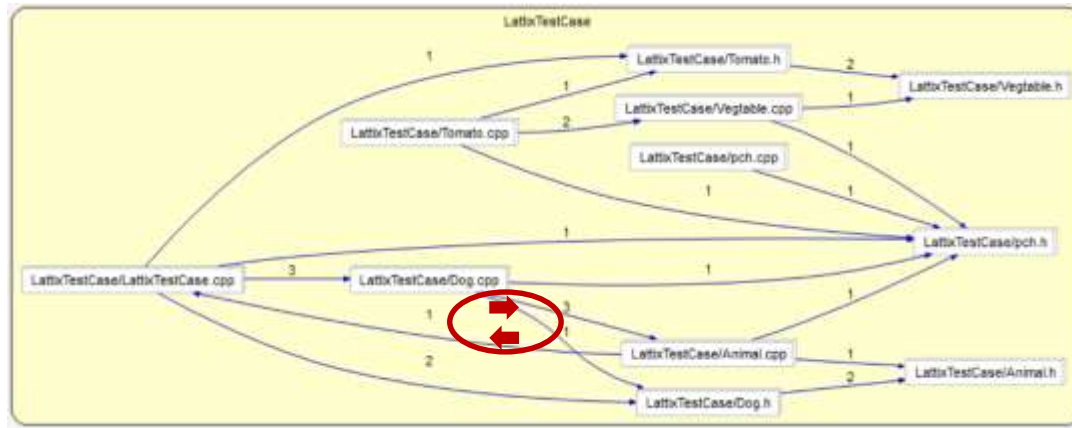


Adapted from [15]

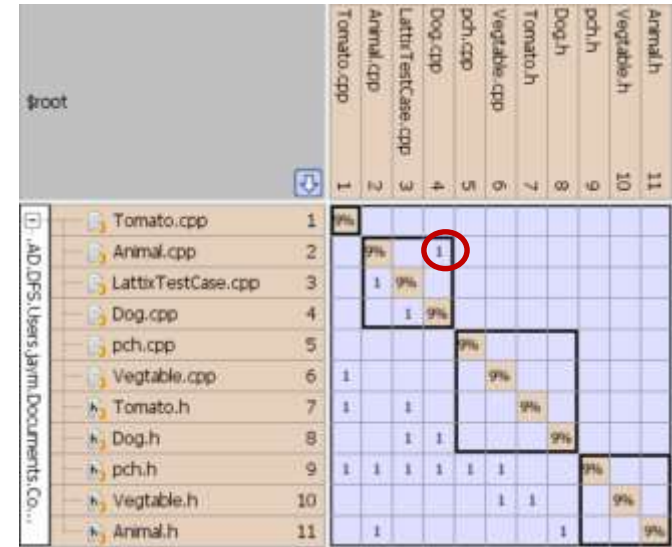
```

graph TD
    A((Module A)) --> B((Module B))
    B --> C((Module C))
    C --> A
  
```

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



Dependency Graph – Created Using SciTools Understand

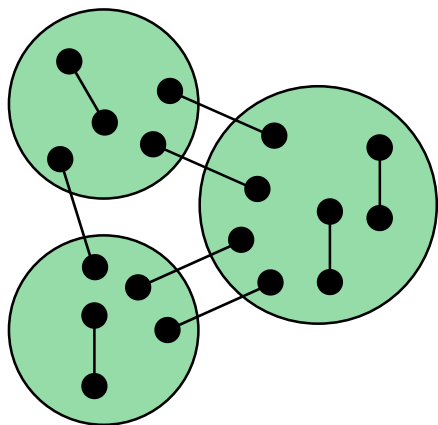


DSM – Created Using Lattix Architect

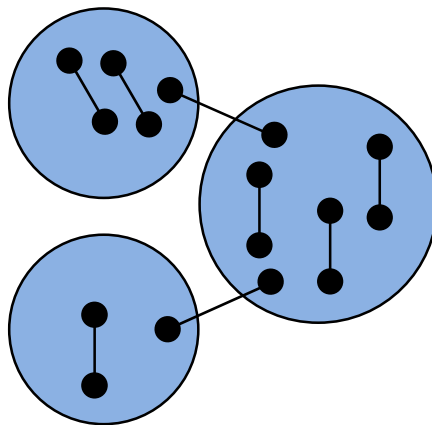
Two Dependency Views of the Same System containing a Cycle

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 modify and maintain?

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

Software Quality - 2

In this section, we'll cover these topics with respect to software quality:

- Architecture
- ▶ • Code quality issues commonly detected using static analysis
- Dynamic analysis: verifying code *in vivo*

Code Quality Issues: Typical Findings

Detected by Tools		Detected by Analyst
Magic numbers	Ignored return values	Watchdog-timer use poor or lacking
Uninitialized variables	Assembly language	Homegrown RTOS or database
Dead code	Conditional compilation overuse	Timing / memory not instrumented
Commented-out code	High complexity code	Poor error handling / logging
Vulnerable library calls	Cyclic dependencies	CPU, OS, or database dependencies
Unvalidated inputs	Build warnings	Poor concurrency management
Global variables	Poor modularity / high coupling	Ad-hoc / incoherent architecture
Stylistic inconsistency	Duplicated code	
Lack of commenting	High SLOC functions and files	

Code Quality Issues: Examples

```

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

Software Quality - 2

In this section, we'll cover these topics with respect to software quality:

- Architecture
- Code quality issues commonly detected using static analysis
- ▶ • Dynamic analysis: verifying code *in vivo*

Dynamic Analysis: Verifying Code *in vivo*

Dynamic analysis (DA) is the analysis of software while it is executing on one or more processors or virtual machines.

Dynamic Analysis validates executable software via two primary means:

Testing

Monitoring

- **Testing** seeks to reveal latent design and coding defects:
 - Preferably as early in the development process as possible.
 - Software testing covers unit, integration, system, and acceptance testing.
- **Monitoring** utilizes field & production inputs rather than tester-selected inputs to:
 - Validate performance, system health, security, licensing, and customer utilization.

TAKEAWAY

- Dynamic Analysis encompasses software testing and run-time monitoring.

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 often 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 inputs.
- UTs can form the basis of regression test suites to validate new system builds.



Key Idea: 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.

TAKEAWAY

- UT is key to code quality and maintenance. Without UT you're developing legacy code [13].
- Best practice: Code passes UT before VCS commit, and UTs are committed to VCS also.

Test Coverage – Structural Test Criteria

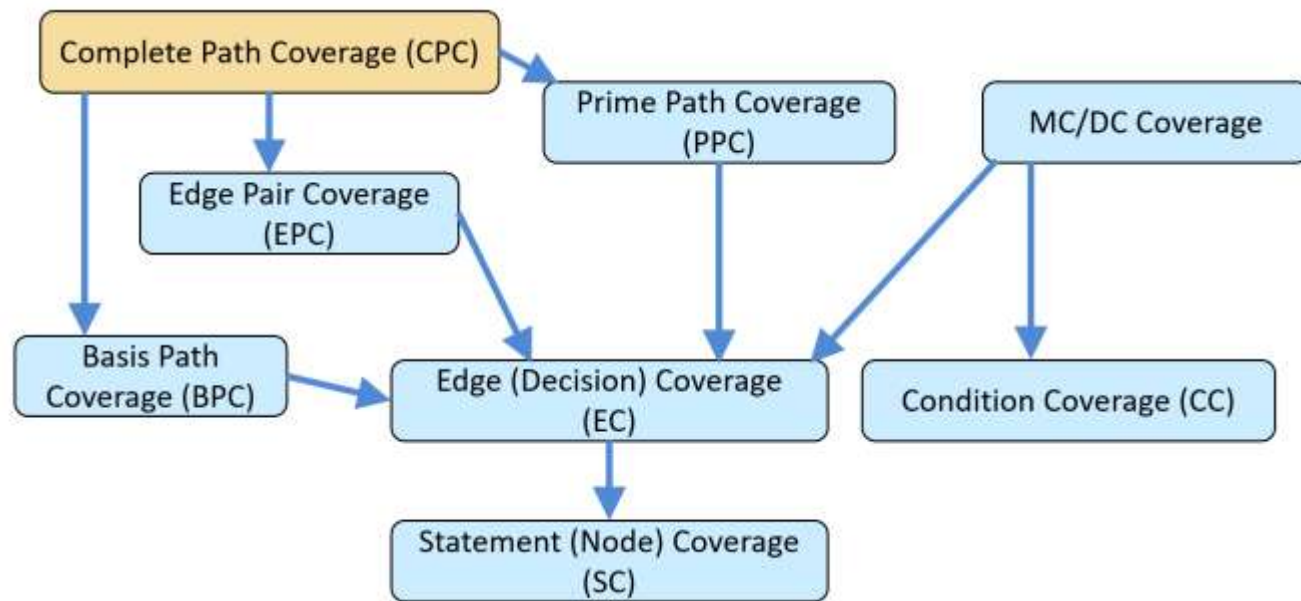
Structural Test Criteria (STC) use Control Flow Graphs (CFG's) to establish a set of Test Requirements (TR's).

- STC's provide implementation-based white-box verification and act as an important adjunct to requirements-based verification, especially for critical systems [16].
- STC's help to quantify the extent of the software verification process and provide an exit criteria for test completion.
- A number of STC's find common use for driving TR's in software testing, for example:
 - Statement coverage
 - Branch coverage
 - Basis-path coverage
 - Also known as Structured Testing where # of test cases req'd = Cyclomatic Complexity
 - Modified condition decision coverage (MC/DC)

Test Coverage – Subsumption

Structural testing criteria (STC) are often related to each other via subsumption.

A coverage criterion A **subsumes** criterion B if and only if every test set that satisfies criterion A also satisfies B.



Knowing where an STC resides within a subsumption chain helps to understand trade-offs between:

- **Adequacy** for uncovering defects, versus
- **Effort** required for test implementation

Test Coverage – Example of Statement (Node) Coverage

Statement Coverage (SC) Test Requirements:

- Execute the statements in the program

Coverage = # statements executed / Total # of statements in the program

Example:

```

1  int PrintSum (int a, int b) {
2      int c = a + b;
3      if (c >= 0)
4          printcolored (GREEN, c);
5      else
6          printcolored (BLUE, c);
7      return (c);
8  }
```

Test cases and coverage:

Case	a	b	Lines executed	Coverage %
1	-2	3	1,2,3,4,7,8	75
2	-2	1	1,2,3,5,6,7,8	87.5
Total coverage =				100

SC is commonly used in industry where ~80-90% coverage is typical.

- However SC is a minimal coverage criteria suitable only for non-critical systems.
- For critical systems MC/DC coverage is recommended - or required (DO-178C).

Fuzz Testing – Randomizing Test Inputs

Fuzz Testing automates the injection of randomized input test cases in order to reveal software defects.

Fuzzers are applied to system interfaces and can provide insight into the system's robustness. Fuzzing is:

- A black box testing technique for finding critical bugs and verifying error handling.
- Cheap insurance - it can often be set up with little effort and run for long test periods.
- An excellent way to catch defects and vulnerabilities missed via SA or other testing.

Fuzz Testing is a useful overlay of other testing for validating system reliability.

- Fuzzing your system's interfaces is a brute-force but effective adjunct form of DA.

Software Monitoring

Software monitoring is a means of validating non-behavioral runtime requirements.

Code is instrumented to monitor aspects of interest during normal or stress-test execution.

Monitoring of	Examples
Performance	Identifying CPU, memory, and communication channel utilization, application profiling to find code bottlenecks and optimize program flow, measuring interrupt and thread-switch latencies, data throughput, etc.
System and network health	Determining memory leaks or stack high-water limits, concurrency issues and hung thread or process issues, network connectivity and bandwidth, etc.
Security events	Detecting application buffer overflows, stack-smashing, unauthorized access attempts, or other security vulnerabilities or violations

TAKEAWAY

- Software monitoring validates non-behavioral requirements during execution.

Monitoring – Instrumented Code

Run-time Monitoring is continual test and measurement executing during normal operation or system stress-testing in order to validate non-behavioral requirements.

Run-time monitoring typically involves the addition of instrumentation code including:

- Commercial products for detecting run-time memory issues, e.g. memory leaks, array bounds overruns, corrupted heap or stack, etc.
- Custom code designed and built into the system to capture and store typical and worst-case processor loading, end-user operations, security or licensing events, etc.



Best Practice: Systems with hard real-time requirements often add timing capture code as a permanent software feature for acceptance testing and field monitoring.

TAKEAWAY

- The addition of run-time monitoring is a best practice, especially for embedded, real-time, and critical systems.

Dynamic Analysis: Strengths

- DA can identify defects and vulnerabilities in a run-time environment, whereas SA does not.
- DA may cover analysis of an entire system, including any portions for which you don't have source code, e.g. third-party libraries and COTS components.
- DA can identify defects and vulnerabilities that were SA false negatives, i.e. DA may pick up defects that SA misses.
- Some DA test tools may be run automatically in a continuous integration (CI) pipeline for automated testing.
- Defects and vulnerabilities identified using DA are actual demonstrated defects, whereas SA defects may or may not manifest in a runtime environment.



Dynamic Analysis: Limitations



- DA testing requires the use of structured test cases, code test harnesses, interface mocks, communication channel sniffers, etc. – all of which must be designed or procured.
- DA testing coverage can be difficult to determine or guarantee.
- DA tools produce false positives and false negatives just like SA tools.
- DA tools may prove the existence of defects and vulnerabilities, but they do not prove the absence of them. Misunderstanding this can lead to a false sense of security.
- Unlike SA, it can be difficult to trace a defect identified using DA back to a specific cause (or line) within the source code in order to mitigate it.

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

End of Module 3

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module 4: Security, Acquisition, & Course Summary

Jay Marchetti

Agenda:

Understanding Software Architecture, Quality, and Security Through Code Analysis

Module	Duration	Topics	Discussing
1	0:10 0:15 0:20	Introduction Overview Quality	Why Code Analysis? Why does the Program Office Care? Quality Attributes
2	0:10 1:00	Review Software Quality - 1	Coding Standards Static Analysis Metrics
3	0:10 1:00	Review Software Quality - 2	Architecture Common Code Quality Issues Dynamic Analysis
4 	0:10 0:15 0:15 0:15	Review Software Security Software Acquisition Course Summary	Security, Standards, and Testing Acquisition Strategy Considerations: RFP's, Contract Execution, and Deployment Checklist for Acquiring High-Quality Code

Understanding Software Architecture, Quality, and Security Through Code Analysis

Review of Module 3

- We discussed architecture and its aspects typically discerned via code analysis, e.g. dependencies, coupling, and cohesion using SA tools:
 - Dependency Graphs and DSM's are common SA architectural visualizations.
 - But we noted that reengineering software architecture from code is a formidable task.
- We covered some of the code quality issues often uncovered during analysis, such as:
 - Embedded constants, uninitialized variables, overly complex code, dead code, stylistic inconsistencies, unvalidated inputs, copy / pasted code, cyclic dependencies, etc.
- We introduced dynamic analysis (DA): analysis of software while it is executing.
 - Testing uses prescribed inputs to reveal latent defects and includes unit, integration, system, and acceptance testing.
 - In particular, we identified unit testing as key to maintainable & high-quality code.
 - Monitoring uses instrumented code for continual measurements of performance, system health, usage metrics, etc.

Software Security, Acquisition, and Course Summary

In this section, we'll cover these topics:

- 
- Security Analysis
 - Acquisition Considerations of Code Analysis
 - Course Summary

Security as a Quality Attribute

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 many possible QA's?

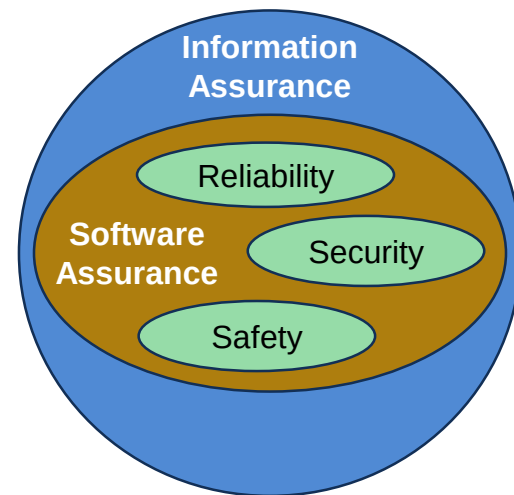
A: It's the only one focused on thwarting an intelligent and motivated adversary...

Security Encompassing QAs – CIA Triad

Software Security is one part, along with software *reliability* and software *safety*, of Software Assurance, which is itself part of Information Assurance.

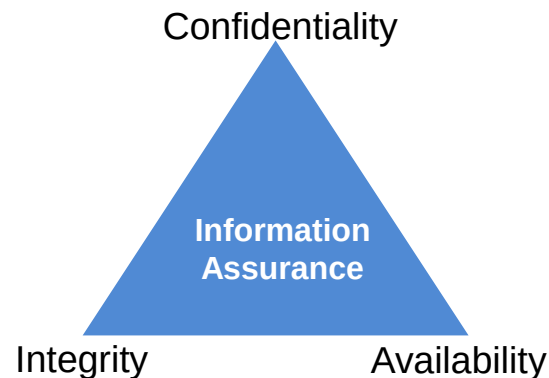
Software Assurance is the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software, throughout the lifecycle.

[Congress.gov – Public Law 112-239 2013 Sec. 933]



Information Assurance – CIA Triad

Confidentiality	Preventing unauthorized access
Integrity	Preventing unauthorized modifications
Availability	Ensuring authorized access



Software Assurance in DoD

DoD Instruction 5000.02 - Operation of the Defense Acquisition System

- The Program Protection Plan (PPP) must describe the plan to apply countermeasures to mitigate associated risks.
- Countermeasures:
 - Include security, secure system design, supply chain risk management, and software assurance.
 - Mitigate or remediate vulnerabilities throughout the product lifecycle, from design through disposal.
- Program Managers must:
 - Implement the use of automated software vulnerability detection and analysis tools.
 - Ensure risk-based remediation of vulnerabilities is addressed in PPPs, included in contract requirements, and verified through continued use of tools and testing.
- In accordance with DoD Instruction 8510.01 (RMF), all programs must have security controls implemented consistent with their information and system categorization.



Software Assurance – Risk Management Framework

Risk Management Framework (RMF)

- Defined by NIST SP 800-37
- Required by DoDI 8510.01, *Risk Management Framework (RMF) for DoD Information Technology*
- Security and Privacy controls defined by NIST SP 800-53

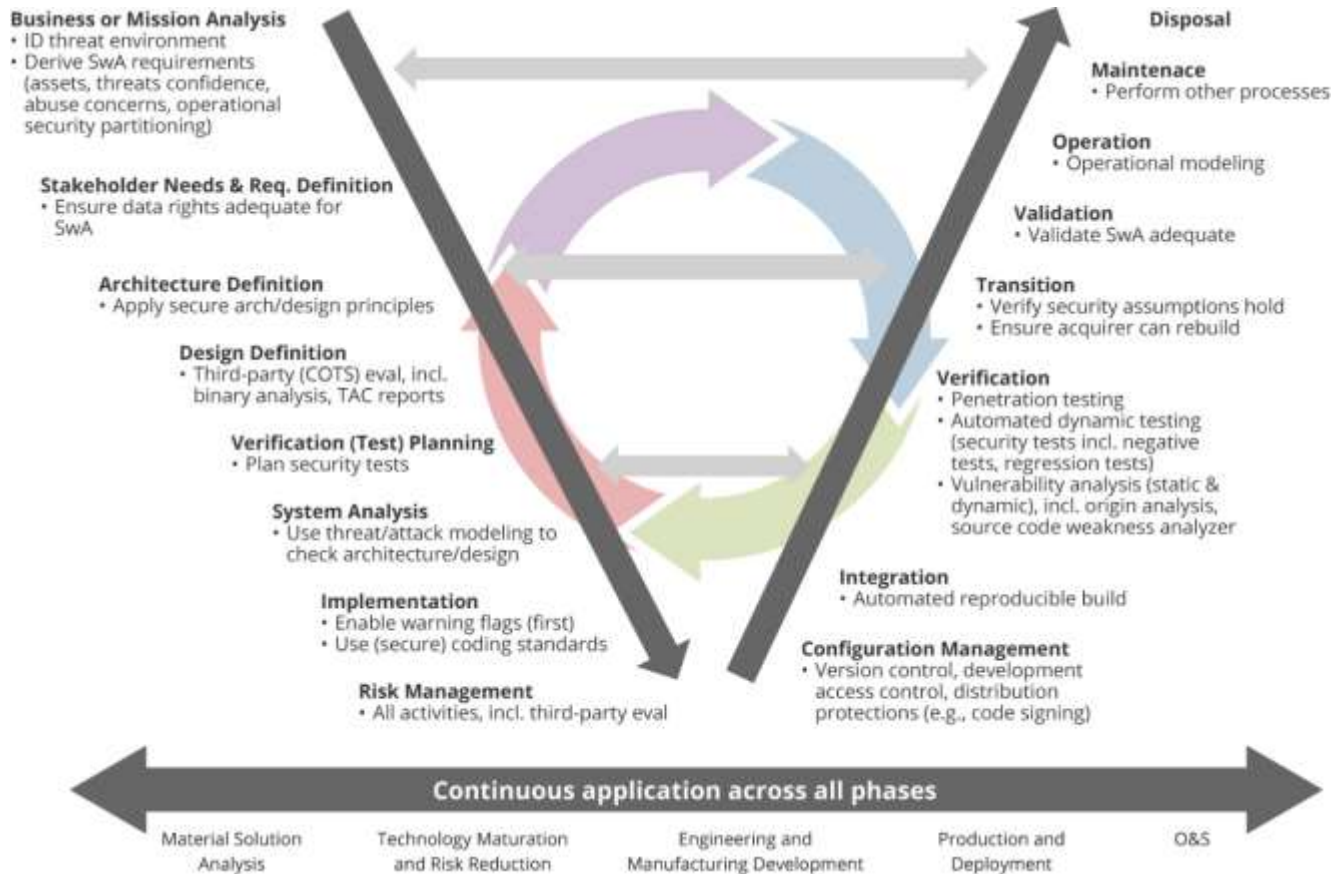
NIST 800-53 specifies Security and Privacy controls to be designed and implemented

- Vulnerability Scanning (RA-5)
- Developer Security Testing and Evaluation (SA-11)
- Flaw Remediation (SI-2)

TAKEAWAY

- RMF includes software assurance and code analysis controls as well as operational controls.

Software Security – Across the Lifecycle



Derived with permission from a diagram originally developed by the DASE (SE)

Secure Coding Standards



MITRE Common Weakness Enumeration (CWE)

- Community-developed list of software & hardware security weakness types
- Over 800, generally language agnostic



Open Web Application Security Project (OWASP)

- OWASP Top 10: most critical security risks to web applications
- Software Assurance Maturity Model (SAMM) framework for software security strategy

Carnegie Mellon University
Software Engineering Institute

SEI CERT Secure Coding Standards

- Language-specific guides for C, C++, Java (including Android), and Perl

SEI CERT Secure Coding Standards

Risk Assessment of Vulnerabilities



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
3	high	run arbitrary code

Likelihood: How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	unlikely
2	probable
3	likely

Cost: The cost of mitigating the vulnerability.

Value	Meaning	Detection	Correction
1	high	manual	manual
2	medium	automatic	manual
3	low	automatic	automatic

SEI CERT Secure Coding Standards

Levels & Mitigation Priorities



Software Engineering Institute

Mitigation Priority (P) = Severity * Likelihood * Cost

High severity, likely,
inexpensive to repair
flaws

Medium severity,
probable, medium
cost to repair flaws

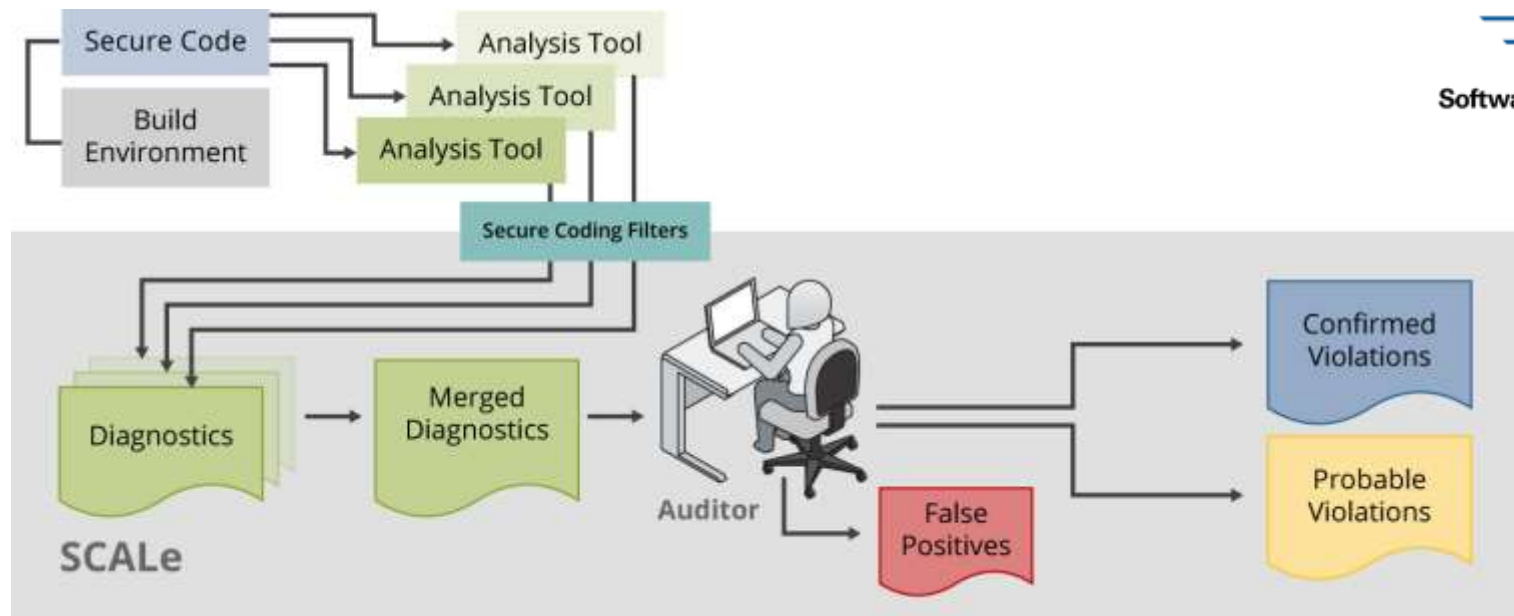
Low severity, unlikely,
expensive to repair flaws

L1: P12 – P27

L2: P6 – P9

L3: P1 – P4

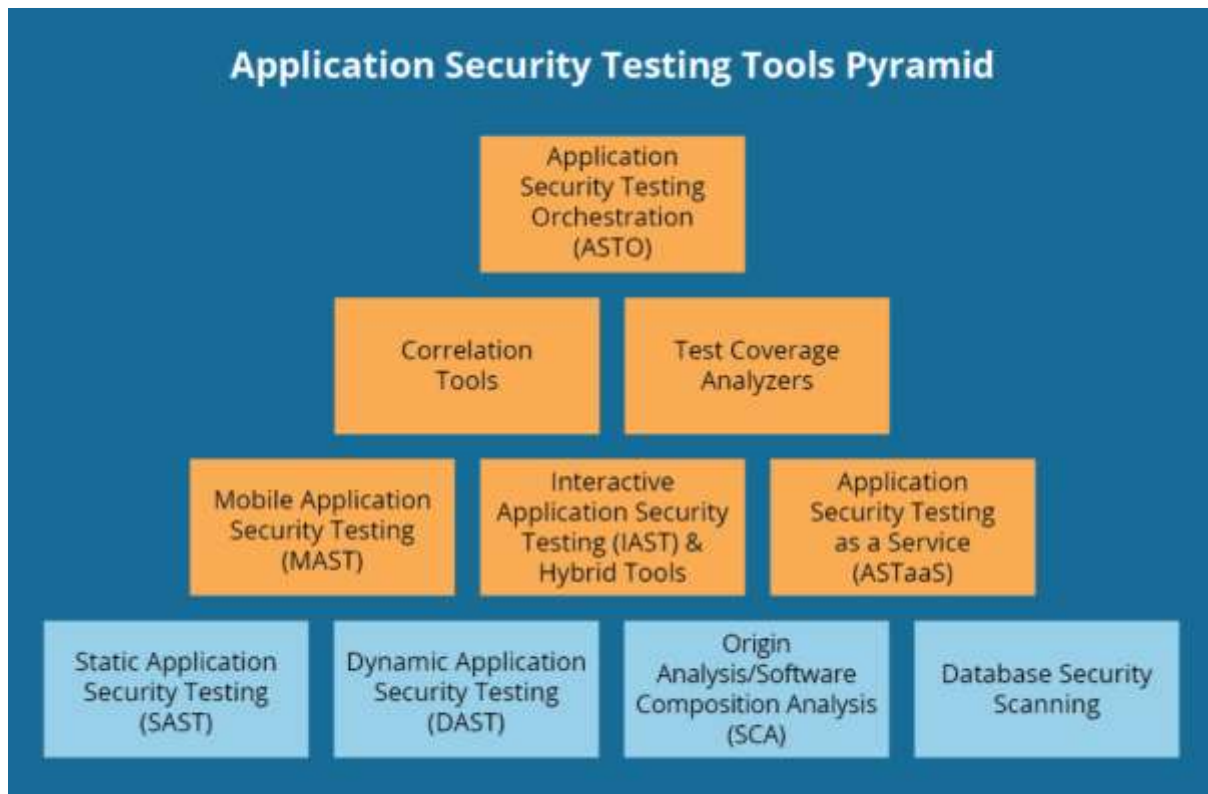
SCALE Secure Coding Conformance Process



SEI CERT Division's Source Code Analysis Laboratory (SCALE) offers conformance testing of C and Java language software systems against the CERT C/C++ Secure Coding Standard and the CERT Oracle Secure Coding Standard for Java.

Application Security Testing (AST) Tools

From the July 2018 SEI Blog “10 Types of Application Security Testing Tools: When and How to Use Them” by Thomas Scanlon.
https://insights.sei.cmu.edu/sei_blog/2018/07/10-types-of-application-security-testing-tools-when-and-how-to-use-them.html



These are the most mature AST tools for addressing common security weaknesses: Start with them.

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

Acquisition Considerations

Acquisition Strategy - 1

Software is pervasive in your system and is being changed constantly.

- Modern practice utilizes Continuous Integration (CI) and deployment throughout the lifecycle accordingly.

Code quality matters. Quality can and should be built-in from the start.

- Have stakeholders agree on QAs before issuing the request for proposal (RFP).
- Focus on defect discovery early in the development process, while they are significantly (10-100x) less expensive to fix.

Be proactive. Consider a contractor incentive for code quality, e.g.:

- Performance incentive based on defects discovered in the first N months of operation.

TAKEAWAY

- Build quality into your software system beginning with the acquisition strategy.

Acquisition Strategy - 2

Decide and designate - Who will conduct code analyses?

- Your contractor.
- Government organization.
- 3rd-party hired for independent verification and validation (V&V).

Decide and designate - What software data products and data rights are required?

- Source code.
- Scripts, makefiles, and support files required to build the system.
- Unit test code, test scripts, and test results.
- Code metrics.
- Architecture and design documents.
- Executable software.

RFP Considerations – Performance Work Statement

Require a viable software quality metrics program.

- Require regular reporting of software metrics, e.g. SLOC, complexity, defect burn-down.
- Select metrics meaningful to your program - use the data in making decisions.
- Ensure the program office and contractor have a shared understanding of the metrics.

Require routine code analysis.

- Code analysis should be utilized at multiple points throughout the software development process and use differing analysis tools.
- Work with the prime to plan how the government will evaluate software quality.
- Make sure this requirement flows down to any sub-contractors who are writing code.

Require regular delivery of any software data products you need to independently validate code quality.

RFP Considerations – Section L

You care how your software will be developed.

- Request a description of software architecture based upon your defined QAs.
- Ask for a description of the software development pipeline including:
 - Offeror's coding standards, unique development practices, and how the standards will be enforced.
 - A list of the analysis tools that will be used, e.g. SA, DA, etc.
 - Metrics collected from previous programs and how they reflected software quality.

You care about the provenance of the proposed off-the-shelf and FOSS software:

- Ask for a list of all such products to be used along with their origin information.
- Ask for findings from code analysis on such products conducted by (or for) the offeror.
- Ask for the offeror's plan to contain any unforeseen quality or supply chain issues.

RFP Considerations – Section M

Evaluate the requested software architecture description.

Evaluate the offeror's software development pipelines:

- Are there different pipelines for different kinds of code?
- At what points in the pipeline will static and dynamic code analysis be conducted?
- What is the offeror's experience level in using this pipeline? These tools?
- Does the offeror have software quality metrics to share from similar programs?
- How will the offeror manage multi-developer version control? What VCS tools and merge strategy will be used?

Consider using 3rd-party code analysis to validate the offeror's assumptions regarding levels of code reuse.

Contract Execution Considerations

Modern software development practice:

- Uses an Agile / DevOps / Continuous Integration and Continuous Deployment (CI/CD) process with code SA, build, and regression testing done nightly.
 - Production quality code is completed every 2 to 4 weeks.
- Ensures that the contractor is using regular in-process SA to continuously enforce coding standards and assess the code quality.
- Ensures that the software is instrumented for DA, e.g. memory and timing.
- Evaluates the as-built architecture regularly to ensure that the implementation conforms to the planned architecture.
- Collects and uses software quality metrics from Day 1.

TAKEAWAY

- Do not wait until delivery to verify your software's quality.

Software Engineering Institute

Understanding Software Architecture, Quality, and Security Through Code Analysis

Course Summary

Understanding Software Architecture, Quality, and Security Through Code Analysis - Key Ideas



IDEA	TAKEAWAY
Quality attributes	Document the system and software QAs important to your stakeholders. Use code analysis to verify that the code actually embodies them.
Architecture	Every system has an architecture. Use code analysis to verify legacy and new systems' architectures for essential QAs and for effective maintenance and reuse efforts.
Standards	Someone will likely be modifying this code for a very long time. Use code analysis to verify appropriate coding standards are used throughout the code's development.
Start early	For DoD systems, realize "the code <i>will</i> be scanned." Be proactive and insist on static and dynamic analyses early in the software development lifecycle.
	• Verify code using static analysis tools as an integral part of development.
	• Validate the system via dynamic analysis from unit through acceptance testing.

Checklist for Acquiring High-Quality Code



The following slides provide a checklist for acquirers engaging contractors for a new project.

Some checklist items relate to topics that are beyond code analysis and were not covered in this presentation, but they have negatively impacted some past programs we have seen. These are included in the spirit of completeness.

Checklist for Acquiring High-Quality Code (1)



✓	Item
	Document / approve requirements for behavioral as well as non-behavioral QAs over the SLC.
	Document / approve any safety- and/or mission-criticality requirements for the system up front.
	Specify data rights for source code, test code, and build files at any time upon PMO request.
	Request / approve a copy of the contractor's stylistic coding standard for use by developers.
	Document / approve any critical system coding standards for use by developers.
	Request / approve a list of all static analysis tools that the contractor will use.
	Request that a report of basic code metrics be supplied monthly to the PMO for review.
	Request / approve the version control tool(s) and branching strategy that the contractor will use.
	Request / approve the defect tracking tool(s) that the contractor will use.
	Request that a report of defect trends be supplied monthly to the PMO for review.

Checklist for Acquiring High-Quality Code (2)



✓	Item
	Request / approve a list of all real-time and other OSs and development tools to be used.
	Request / approve contractor's planned use / level of unit testing during development.
	Request / approve contractor's planned use / level of build automation for the system.
	Request / approve contractor's planned use / level of test automation for the system.
	Request / approve a copy of the contractor's deployment plan for software releases and updates.
	Request / approve how requirements and capabilities will be traceable down to the code.
	Document / approve unambiguous acceptance criteria for final code w.r.t. any remaining defects.
	Document / approve a means for code security and safety certification audits before delivery.
	Apply items in this checklist for firmware as well if the system employs FPGAs.
	Ensure that the items in this checklist are flowed down and applied to all subcontractors.

Adjourn

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

Software Engineering Institute

4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Email: info@sei.cmu.edu

Phone: +1 412-268-5800

Fax: +1 412-268-6257

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.