
Testing

Case Study 3

In May 2015, Airbus issued an alert to urgently check its A400M aircraft when a report detected a software bug that had caused a fatal crash earlier in Spain. Before this alert, a test flight in Seville caused the death of four Air Force crew members, and two were left injure

DoD DevSecOps Testing

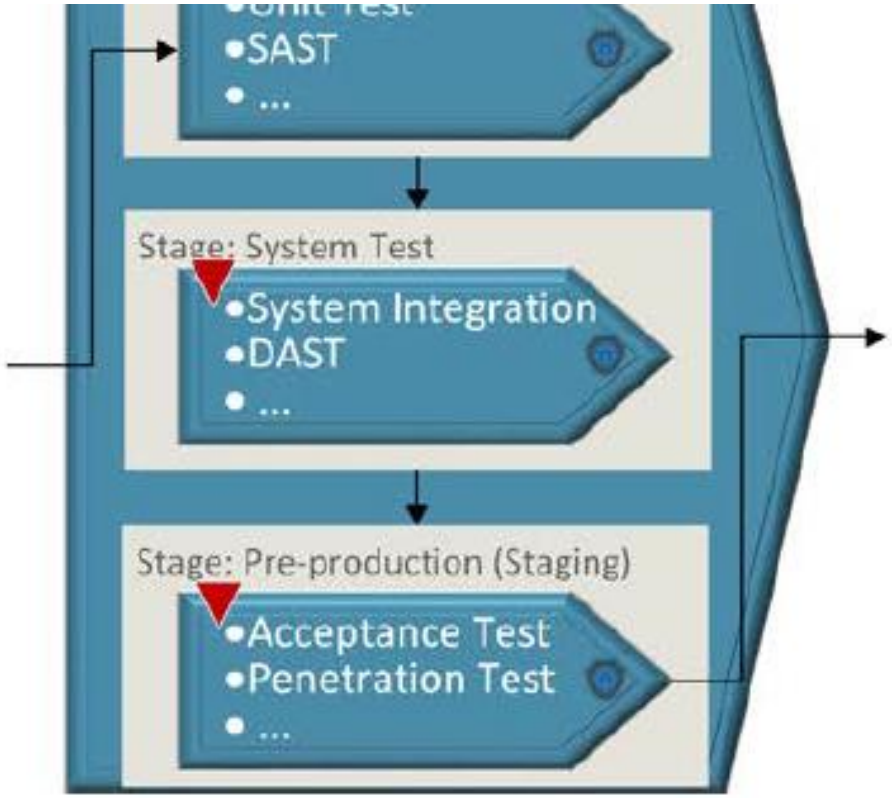


Figure 5: Application DevSecOps Processes

DoD Enterprise DevSecOps Reference Design August 2019

DoD DevSecOps Testing

- The discipline of testing changes within the automated processes of DevSecOps. Test moves away from the traditional "test the system as implemented" and becomes "test the code that implements the system." One implication of this evolution is that re-skilling of the test team is needed; the old skill set of "sit at a screen and use the app as you were trained for 3 days to use it" is no longer applicable. Rather, testing is about automation, and testers will need to become coders of that automation.

- DevSecOps Fundamentals Guidebook: DevSecOps Tools & Activities March 2021 Version 2.0

Testing Types

SAST - Static Application Security Testing

DAST – Dynamic Application Security Testing

IAST - Interactive application security testing

RASP - Runtime application self-protection

SCA - Software composition analysis

DoD DevSecOps Testing

Development stage: unit test, SAST discussed in the build phase System test stage: DAST or IAST, integration test, system test

Pre-production stage: manual security test, performance test, regression test, acceptance test, container policy enforcement, and compliance scan

DoD DevSecOps Testing

- Common Testing Categories
- • Unit and Functional Testing.
- • Integration Testing.
- • Performance Testing.
- • Interoperability Testing.
- • Deployment Testing (normally conducted in a staging environment).
- • Operational Testing (normally conducted in a production environment).
- • Static Application Security Testing (SAST).
- • Dynamic Application Security Testing (DAST).
- • Interactive Application Security testing (IAST).
- • Runtime Application Self-Protection (RASP).

- DevSecOps Fundamentals Guidebook:DevSecOps Tools & Activities March 2021 Version 2.0

Some definitions

Test	A process of finding errors through the occurrences of faults Another definition: the act of exercising test cases
Test case	Description of a set of inputs along with expected outputs
Test data	Values that instantiate a test case
Test Suite	A collection of test cases targeted to test a set of properties such as user interface functionalities, reliability, performance and so on.
SUT	System (or Software) Under Test
Test Harness	Software along with a set of executable test cases to test a SUT. This is part of an **automated testing process.**
Testing and debugging	Testing is a process of finding the presence of an error • May be performed by several people including the programmer Debugging is a process of finding the source of an error, if it is present • Mostly performed by the programmer

ISO-IEC-IEEE 29919-2
Software and systems engineering – Software testing- Test Processes

The goal of each layer is as follows:

- Organizational test process
  - Defining a process for the creation and maintenance of organizational test specifications, such as organizational test policies, practices, processes, procedures and other assets.
- Test management processes
  - Defining processes that cover the management of testing for a whole project or any test level
  - (e.g. system testing) or test type (e.g. performance testing) within a project (e.g. project test
  - management, system test management, performance test management).
- The test management processes are:
  - test strategy and planning process ;
  - test monitoring and control process ;
  - test completion process
- Dynamic test processes
  - Defining generic processes for performing dynamic testing. Dynamic testing may be performed at a particular test level (e.g. unit, integration, system, and acceptance) or for a particular test type (e.g. performance testing, security testing, and functional testing) within a project.
  - The dynamic test processes are:
    - test design and implementation process ii) test environment and data management process
    - test execution process and
    - test incident reporting process.

ISO-IEC-IEEE 29919-2
Software and systems engineering – Software testing- Test Processes

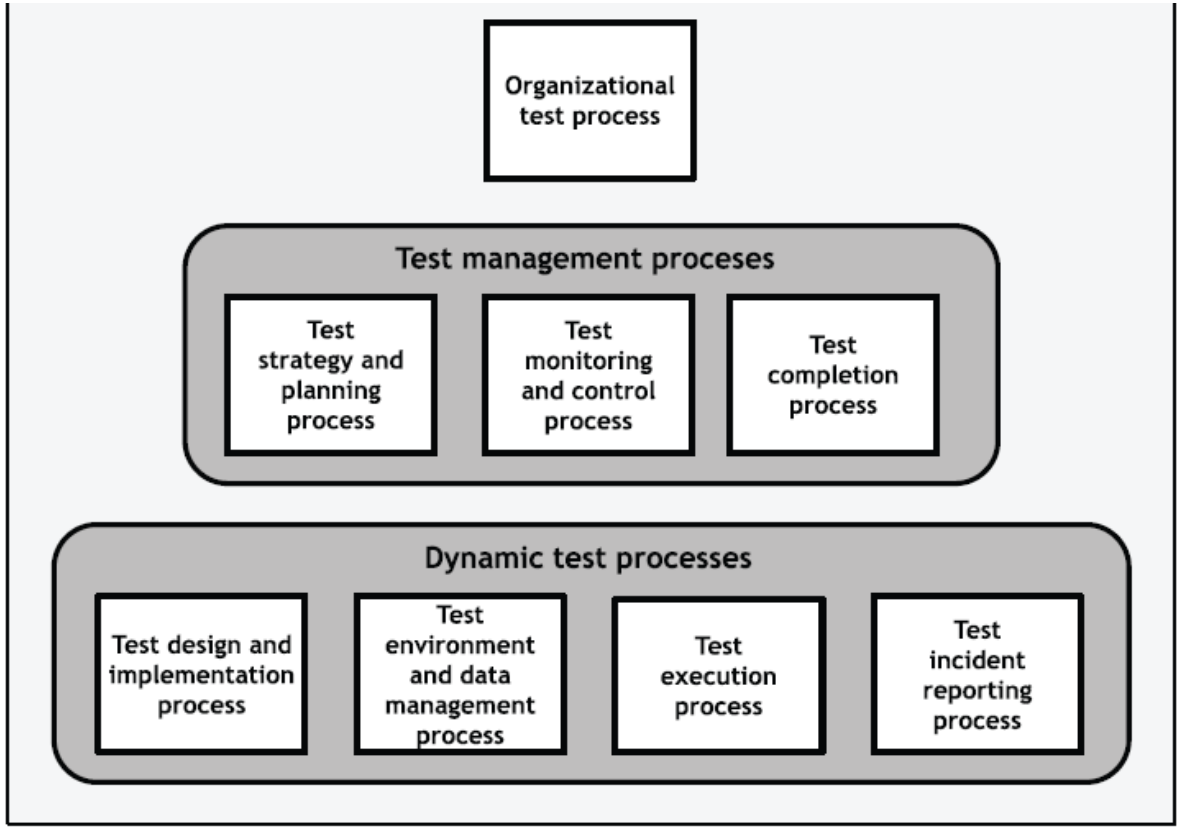


Figure 2 — The multi-layer model showing all test processes

ISO-IEC-IEEE 29919-2
Software and systems engineering – Software testing- Test Processes

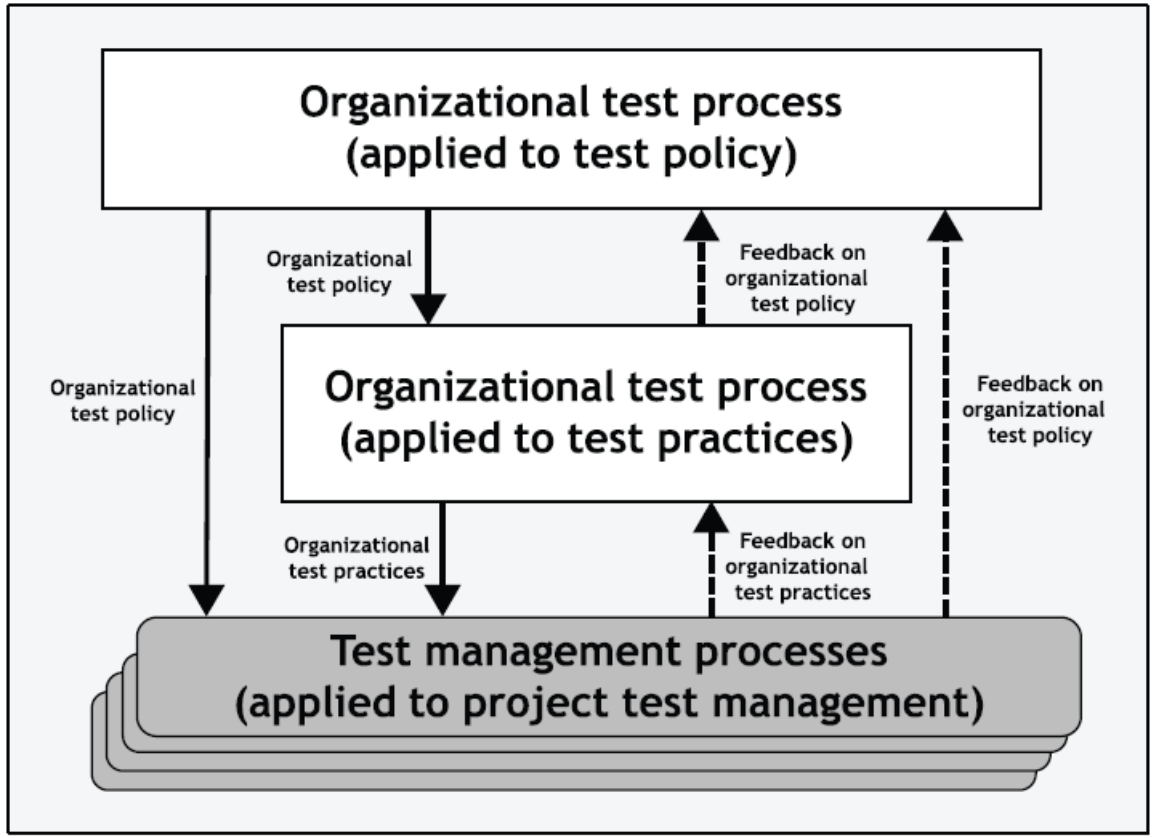


Figure 3 — Example organizational test process implementation

ISO-IEC-IEEE 29919-2 Software and systems engineering – Software testing- Test Processes

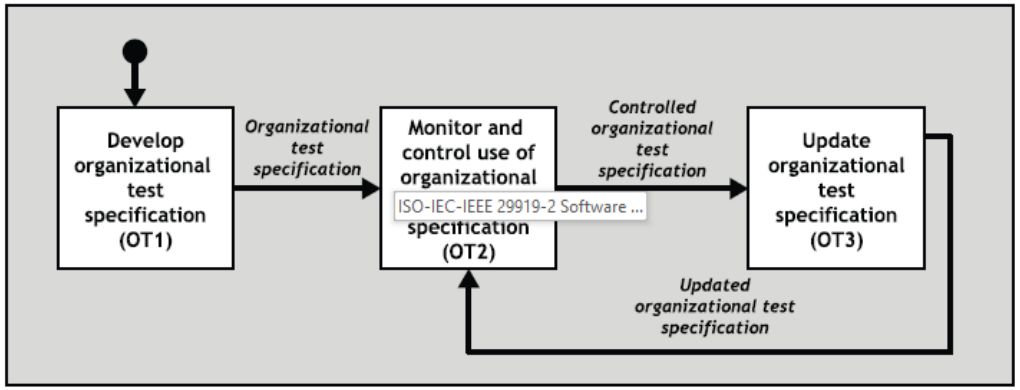


Figure 4 — Organizational test process

ISO-IEC-IEEE 29919-2 Software and systems engineering – Software testing- Test Processes

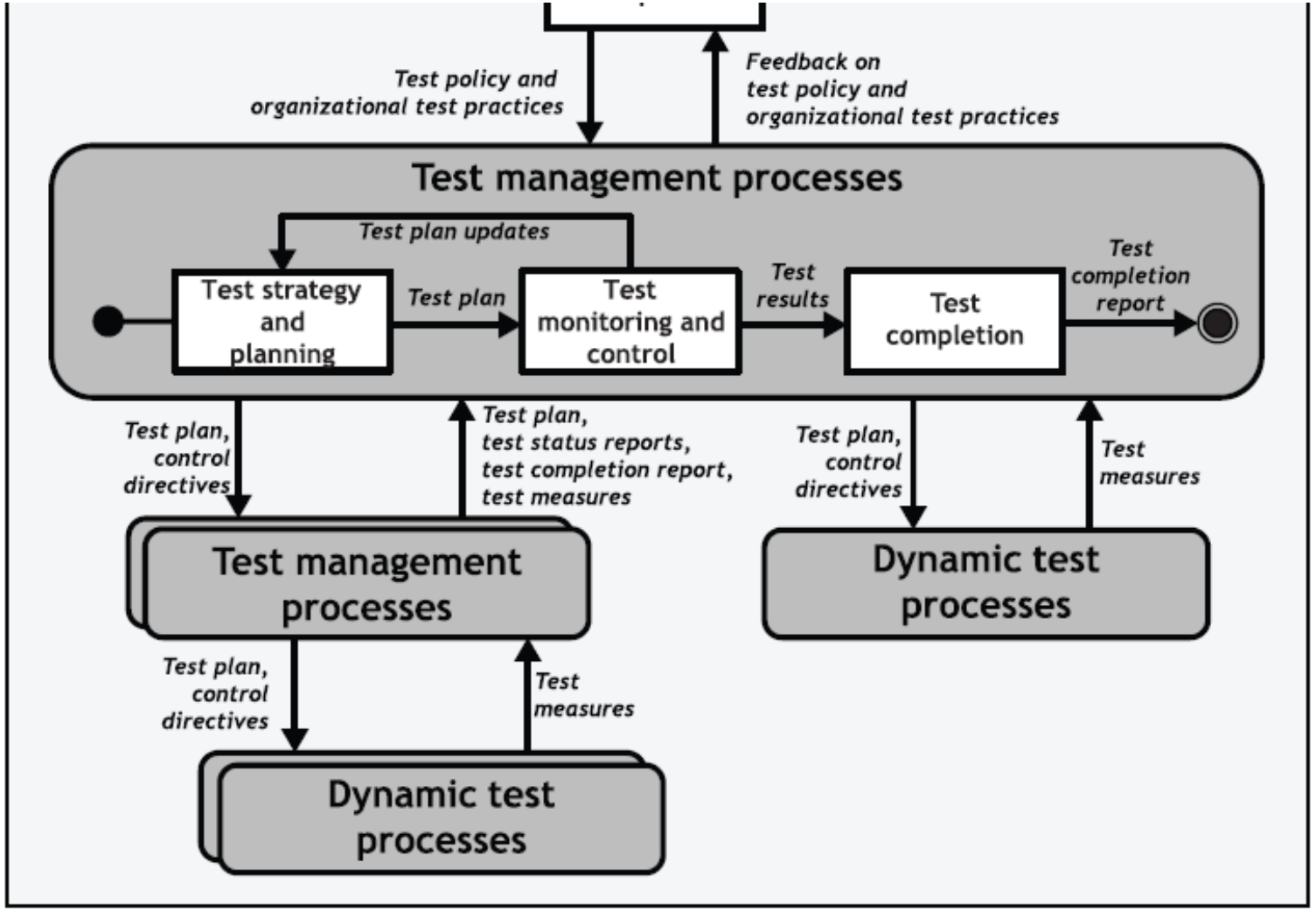


Figure 5 — Example test management process relationships

Software Integrity Levels

Criticality	Description	Level
High	Selected function affects critical performance of the system.	4
Major	Selected function affects important system performance.	3
Moderate	Selected function affects system performance, but workaround strategies can be implemented to compensate for loss of performance.	2
Low	Selected function has noticeable effect on system performance but only creates inconvenience to the user if the function does not perform in accordance with requirements.	1

Selecting Test Tools

- Accomplishment of specified tool objectives
- Ease of use
- Ease of installation
- Minimum processing time
- Compatibility with other tools
- Low purchase or lease cost
- Documentation, Training, and Support availability

Classes of Test Tools

- Automated Regression Testing Tools
- Defect Management Tools
- Performance/Load Testing Tools
- Manual Tools
- Traceability Tools
- Code Coverage
- Test Case Management Tools
- Common tools that are applicable to testing

SMART Goals

- **S**pecific
- **M**easurable
- **A**greed Upon
- **R**ealistic
- **T**ime Frame

Test Planning Vocabulary

- Test Case: Test cases are how the testers validate that a software function meets the software specifications (i.e., expected results).
- Test Data: Test data is information used to build a test case.
- Test Scripts: Test scripts are an online entry of test cases.
- Risk: Risk is the potential loss to an organization.
- Risk Analysis: Risk analysis is an analysis of an organization’s information resources, its existing controls, and its remaining organization and computer system vulnerabilities. It combines the loss potential for each resource or combination.

Test Planning Vocabulary

- Threat: A threat is something capable of exploiting vulnerability.
- Vulnerability: Vulnerability is a design, implementation, or operations flaw that may be exploited by a threat.
- Control: Control is anything that tends to cause the reduction of risk.

Test Planning

- • Test Objectives
- • Acceptance Criteria
- • Assumptions
- • People Issues
- • Constraints

Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process

Boundary value testing

Partition system inputs and outputs into 'equivalence sets'

- If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are < 10,000,
  10,000 - 99, 999 and > 10, 000

Choose test cases at the boundary of these sets

- 00000, 09999, 10000, 99999, 10001

White-box testing

Sometime called structural testing or glass-box testing

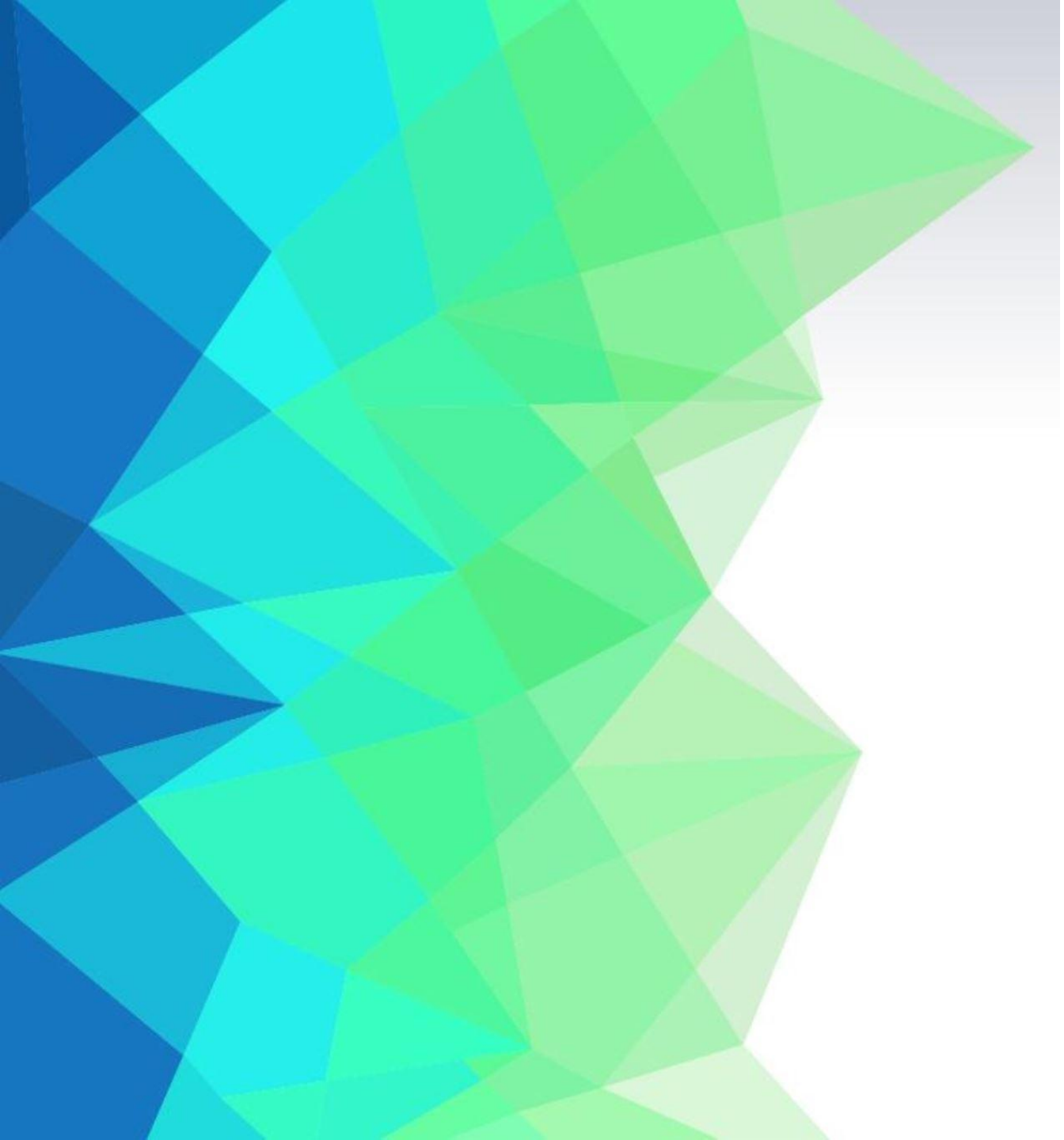
Derivation of test cases according to program structure

Knowledge of the program is used to identify additional test cases

Objective is to exercise all program statements (not all path combinations)

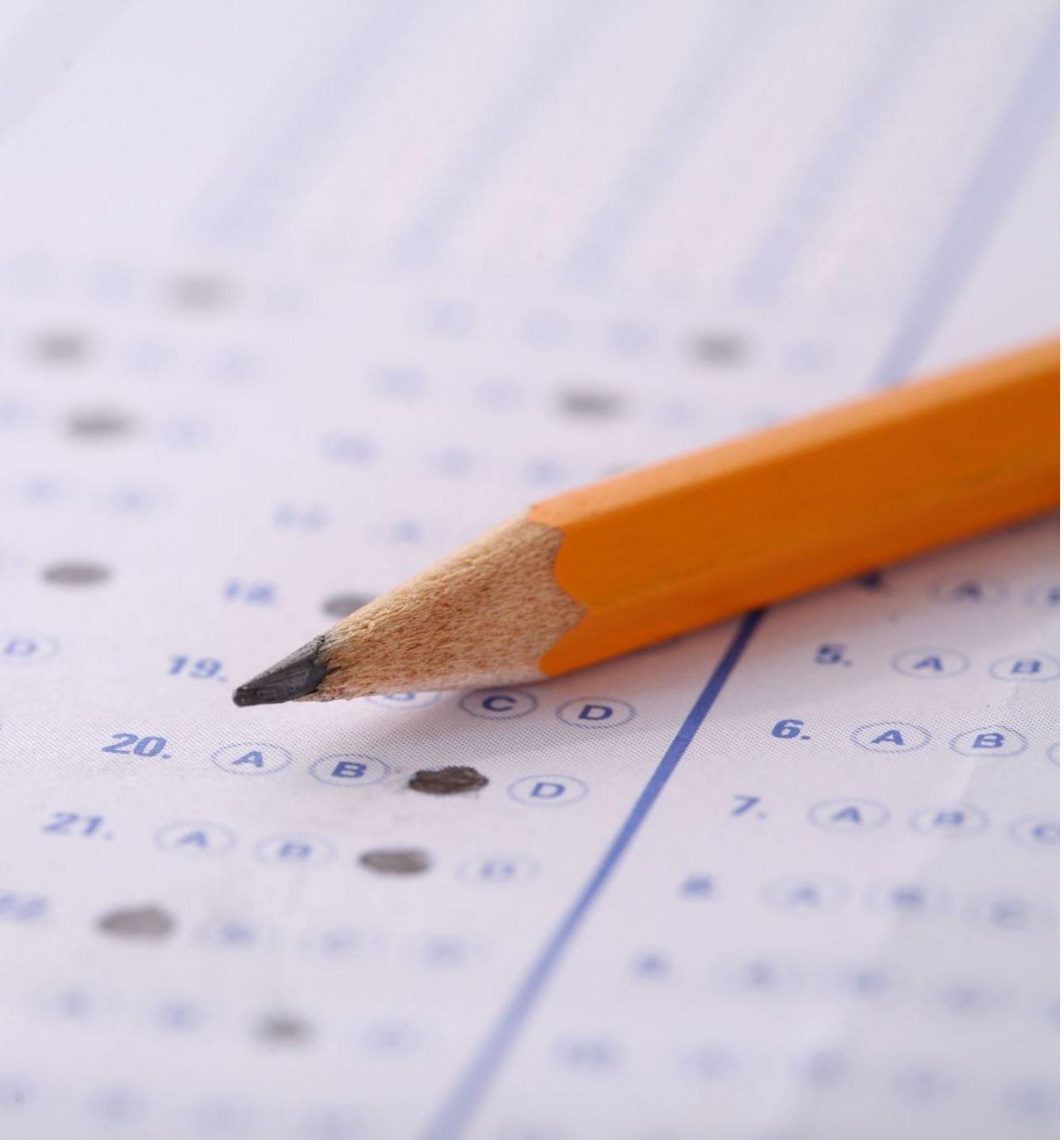
Types of structural testing

Statement coverage -	• Test cases which will execute every statement at least once. • Tools exist for help • No guarantee that all branches are properly tested. Loop exit?
Branch coverage	• All branches are tested once
Path coverage - Restriction of type of paths:	• Linear code sequences • Definition/Use checking (all definition/use paths) • Can locate dead code



Software testing metrics

- Defects rates
- Errors rates
- Number of errors
- Number of errors found per person hours expended
- Measured by:
  - individual
  - module
  - during development
- Errors should be categorized by origin, type, cost

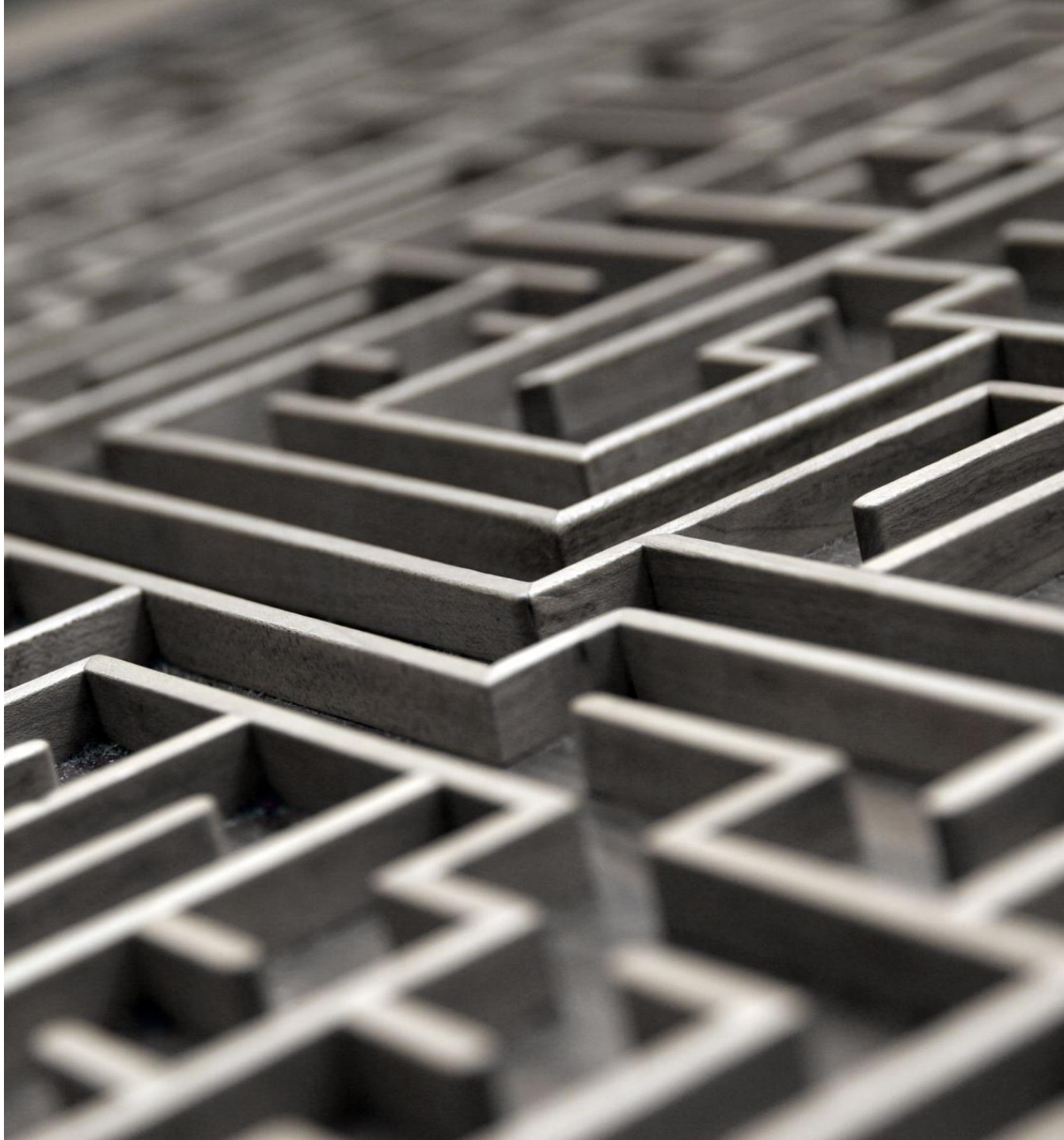


Test Writing First

- Idea is to write tests, where each test adds some degree of functionality
- Passing the tests should indicate working code (to a point)
- The tests will ensure that future changes don’t cause problems

Running Tests

- Use a test harness/testing framework of some sort to run the tests
  - A variety of ways to do this, including many existing frameworks that support unit tests
  - JUnit is the most well-known, but there is similar functionality across a wide range of languages



Test framework

- Specify a test *fixture*
  - Basically, builds a state that can be tested
  - Set up before tests, removed afterward
- Test suite run against each fixture
  - Set of tests (order should not matter) to verify various aspects of functionality
  - Described as series of assertions
- Runs all tests automatically
  - Either passes all, or reports failures
    - Better frameworks give values that caused failure

Checklist: Test Cases

- Does each requirement that applies to the class or routine have its own test case?
- Does each element from the design that applies to the class or routine have its own test case?
- Has each line of code been tested with at least one test case?
- Has this been verified by computing the minimum number of tests necessary to exercise each line of code?
- Have all defined-used data-flow paths been tested with at least one test case?
- Has the code been checked for data-flow patterns that are unlikely to be correct?
- Defined-defined, defined-exited, defined-killed, etc.
- Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past?
- Have all simple boundaries been tested: maximum, minimum, off-by-one?
- Have compound boundaries been tested: combinations of input data that might result in a computed variable that is too small or too large?
- Do test cases check for the wrong kind of data?
- Are representative, middle of the road values tested?
- Are the minimum and maximum normal configurations tested?
- Is compatibility with old data tested?
- Do test cases make hand-checks easy?

Quality perspectives

1. Transcendent – I know it when I see it

2. Product-Based – Possesses desired features

3. User-Based – Fitness for use

4. Development- and Manufacturing-Based – **Conforms to requirements**

5. Value-Based – At an acceptable cost

Quality Control

Quality control identifies defects so they can be corrected.

Quality control relates to a specific product or service.

Quality control verifies whether specific product or service has a specific attribute.

Quality control is the responsibility of the team/worker.

Quality Assurance

Quality assurance sets up measurement programs to evaluate processes.

Quality assurance identifies weaknesses in processes and improves them.

Quality assurance is a management responsibility but can be performed by a staff.

Quality assurance is concerned with all of the products and services, not just an individual product or service.

Quality assurance is sometimes referred to as quality control over quality control because it evaluates whether quality control is working.

Quality assurance personnel should never perform quality control unless it is to validate quality control.

Cost of Quality

- The three categories of costs associated with producing quality products are:
- • **Prevention Costs**
- Money required to create quality processes and products.
- • **Appraisal Costs**
- Money spent to **review completed products against requirements**. Appraisal includes
- the cost of inspections, testing, and reviews.
- • **Failure Costs**
- All costs associated with defective products that have been delivered to the user or moved into production. This includes repair, replacement, support, downtime, and in some case litigation.

Quality Factors

Correctness Extent to which a program satisfies its specifications and fulfills the user's mission objectives.

Reliability Extent to which a program can be expected to perform its intended function with required precision

Efficiency The amount of computing resources and code required by a program to perform a function.

Integrity Extent to which access to software or data by unauthorized persons can be controlled.

Usability Effort required learning, operating, preparing input, and interpreting output of a program

Maintainability Effort required locating and fixing an error in an operational program.

Testability Effort required testing a program to ensure that it performs its intended function.

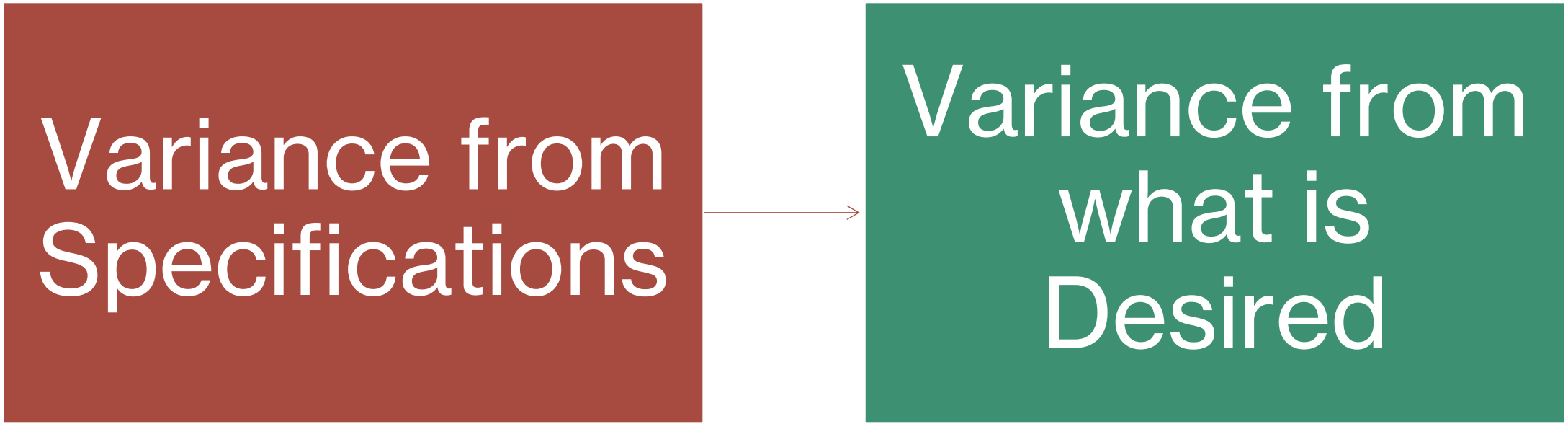
Flexibility Effort required modifying an operational program.

Portability Effort required to transfer software from one configuration to another.

Reusability Extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform.

Interoperability Effort required to couple one system with another

What is Variance?



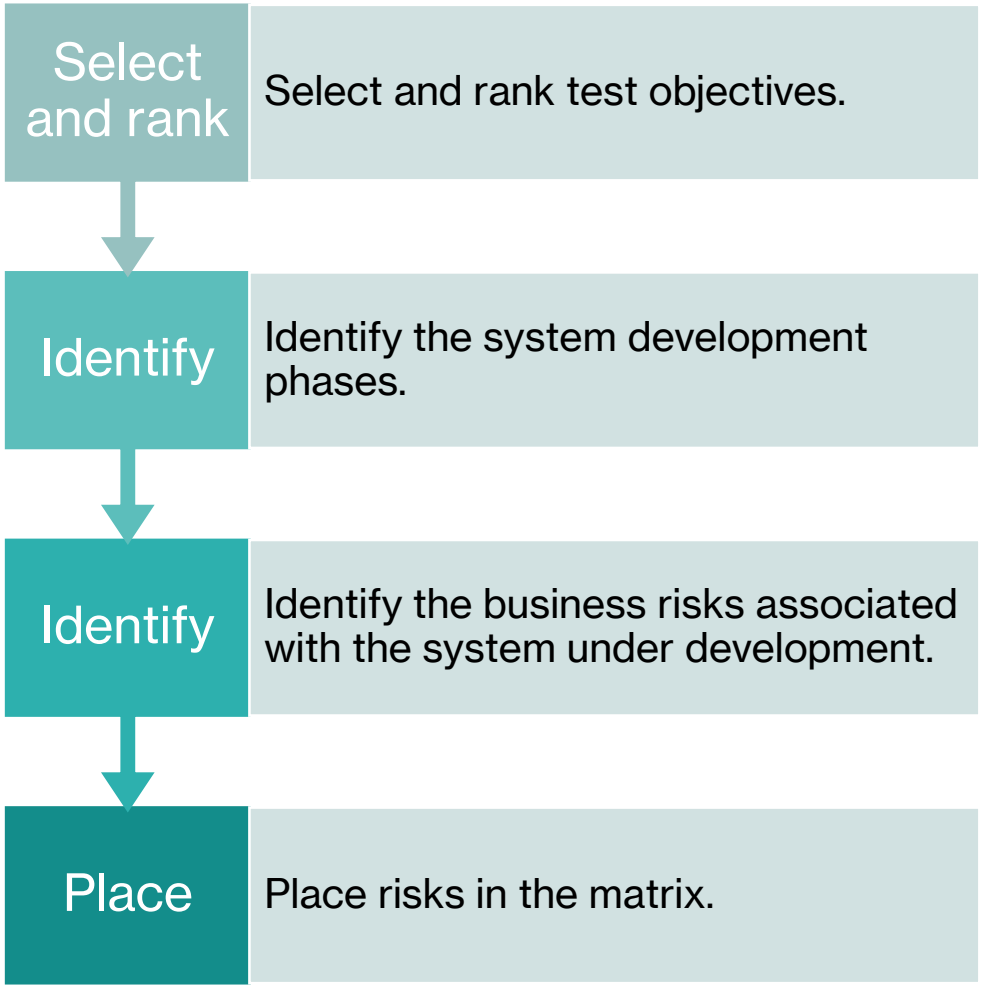
Software Product Defects

Software Design Defects	Designing software with incomplete or erroneous decision-making criteria Failing to program as designed Failure to validate data
Data Defects	Incomplete data used by automated decision-making applications.

Reasons for Software Defects

- IT improperly **interprets requirements**
- Users specify the **wrong requirements**
- **Requirements are incorrectly recorded**
- Design specifications are incorrect
- Errors in program coding
- Data entry errors
- Testing errors
- Tests falsely detect an error
- The corrected condition causes another defect

Test Planning

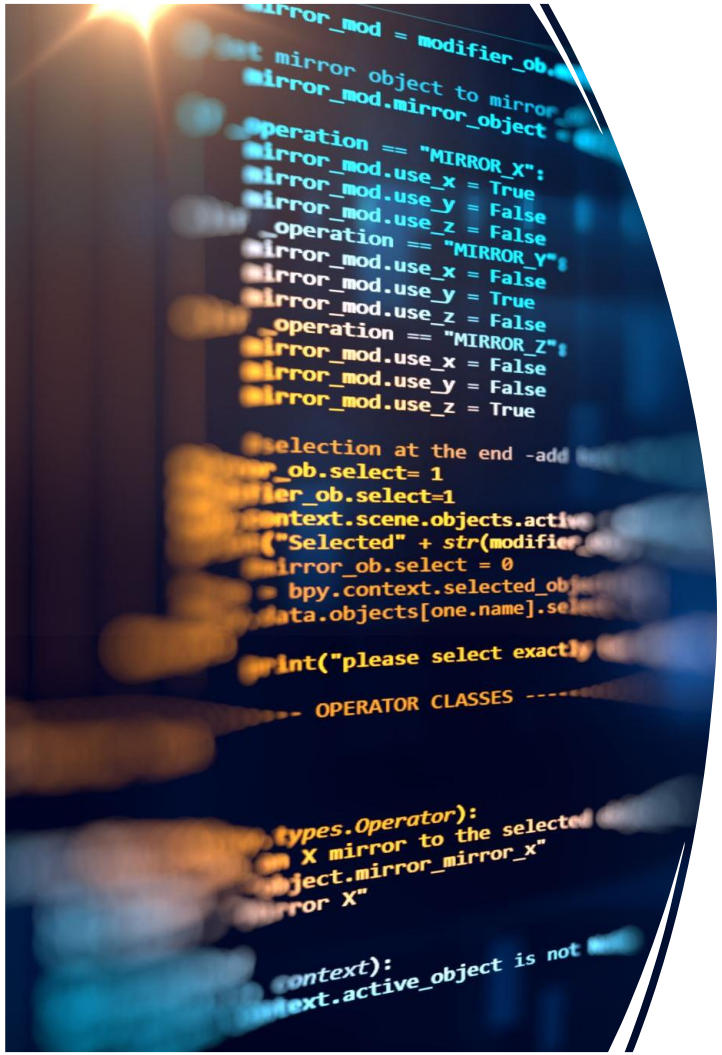


Testing Constraints

- Limited schedule and budget
- **Lacking or poorly written requirements**
- Changes in technology
- Limited tester skills

Source of most problems in testing

- Poorly defined testing objectives
- Testing at the wrong phase in the life cycle
- Ineffective test techniques

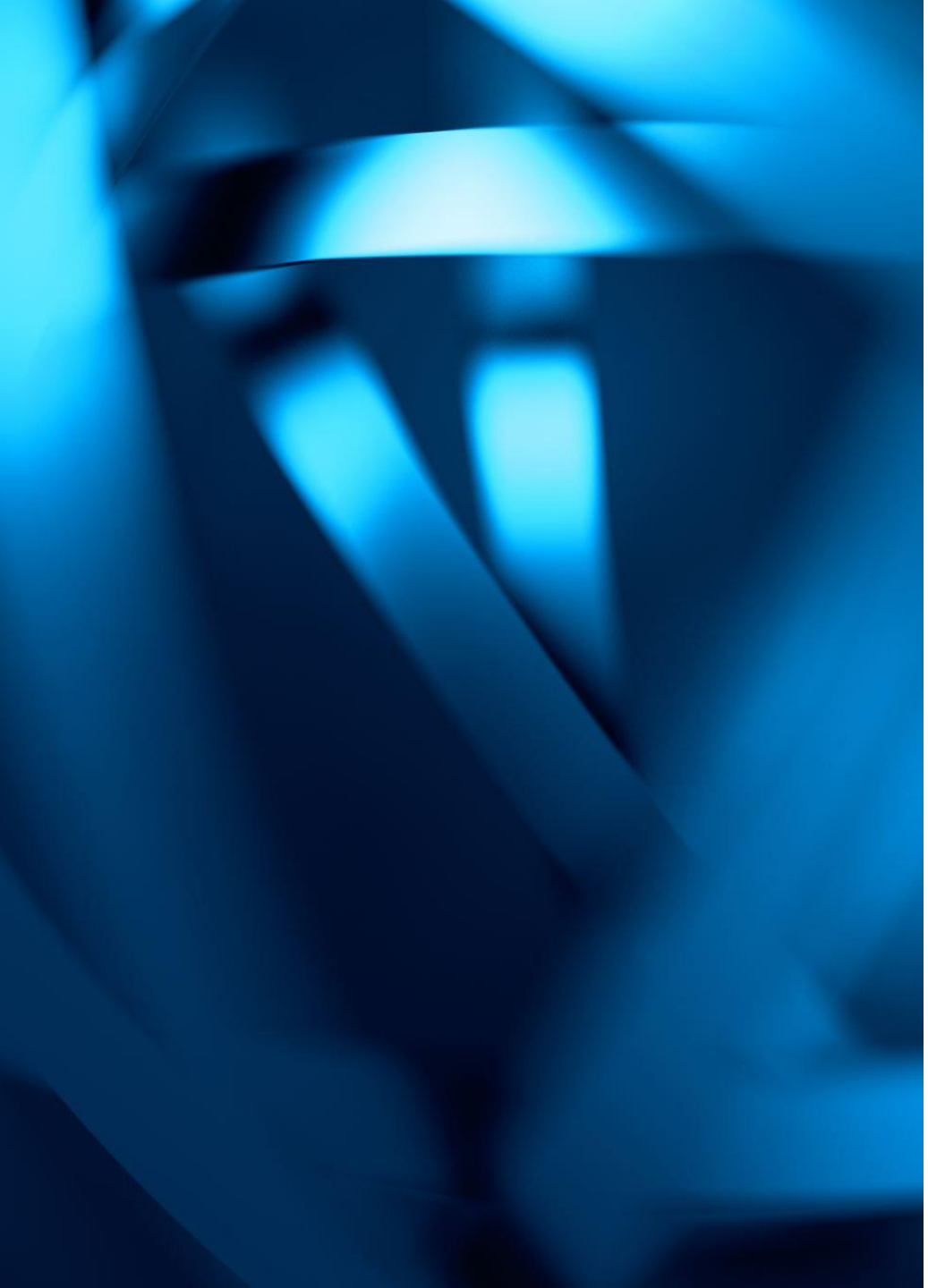


Examples

- Requirements Reviews – Done by developers and users
- Design Reviews - Developers
- Code Walkthroughs – Developers
- Code Inspections - Developers
- Unit Testing – Developers
- Integrated Testing – Developers with assist from independent test team
- System Testing – Independent Test Team
- User Acceptance Testing- users with assist from independent test team

Static vs. Dynamic

- Static testing is performed using the software documentation.
- Dynamic testing requires the code to be in an executable state to perform the tests.
- Most verification techniques are static tests.
- Most validation tests are dynamic tests.



Functional vs Structural

Functional testing is sometimes called black-box testing because no knowledge of the internal logic of the system is used to develop test cases.

Structural testing is sometimes called white-box testing because knowledge of the internal logic of the system is used to develop hypothetical test cases.

Functional Testing Requirements

- Requirements - System performs as specified.
- Regression – Verifies anything unchanged still performs correctly.
- Error Handling – Errors prevented or detected.
- Manual Support – Support process works.
- Inter-system – Data is correctly passed from system to system.
- Control Controls reduce system risk to acceptable level.
- Parallel - Old system and new system are run and the results compared

Structural Testing Techniques

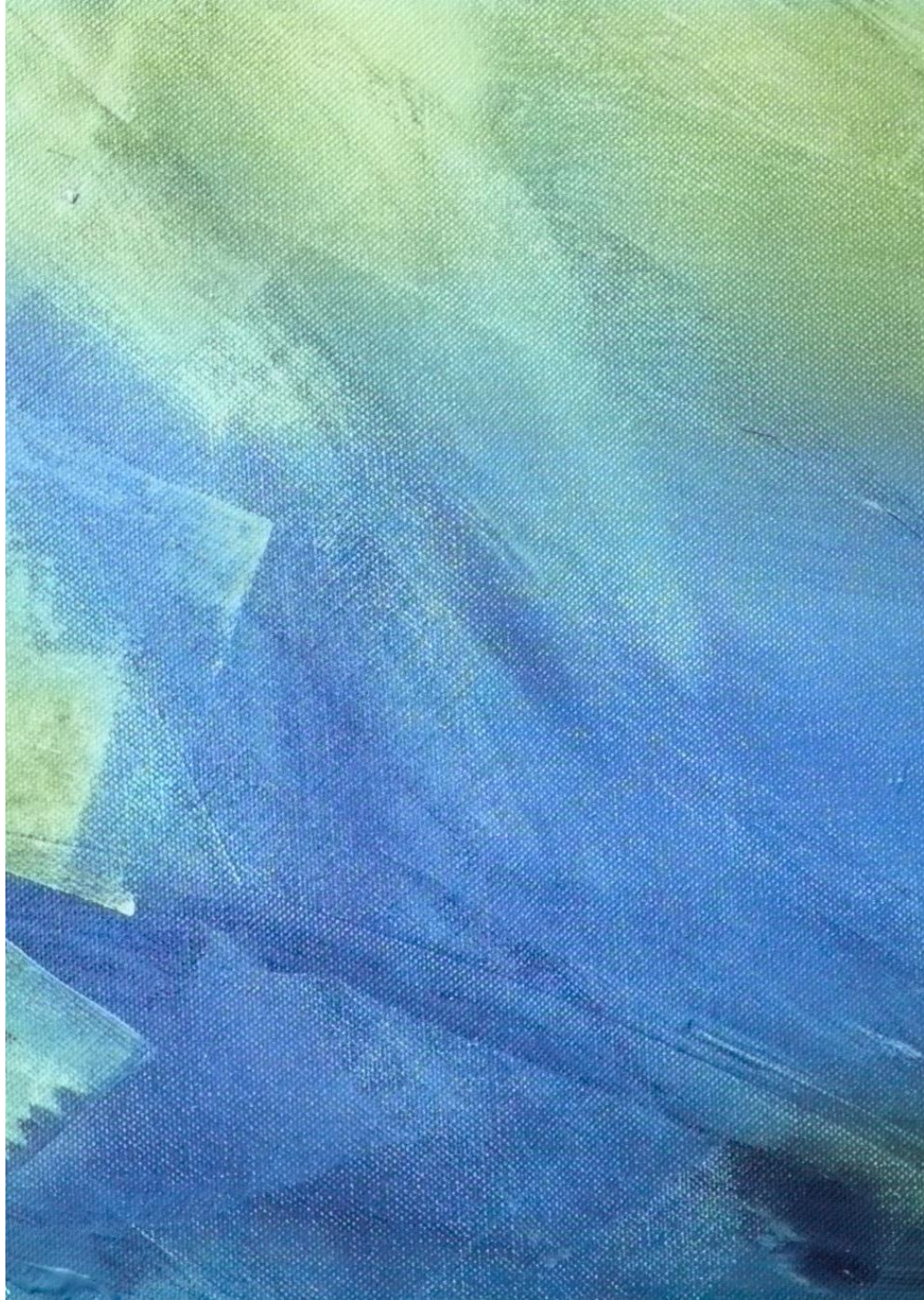
Stress – System can handle heavy load and still produce expected outcomes.

Recovery- System can recover after a failure.

Operations - System can be executed normally.

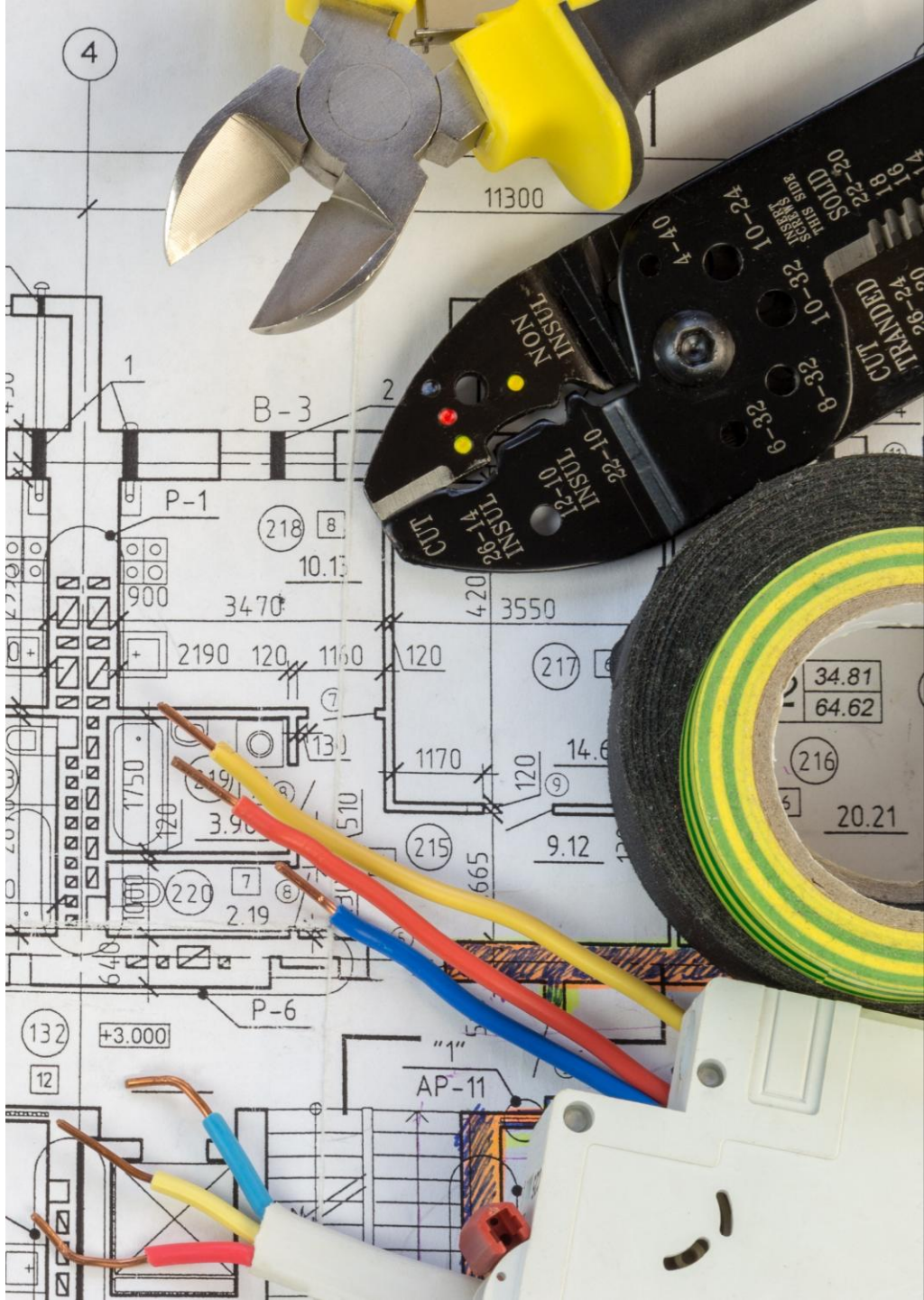
Compliance – System complies with requirements.

Security – System meets security requirements.



Structural Testing

- In structural analysis, programs are analyzed without being executed. The techniques resemble those used in compiler construction.
    - Complexity Measures
    - Data Flow Analysis
    - Symbolic Execution



Structural Testing

Testing	Statement Testing: Statement testing requires that every statement in the program be executed.
Testing	Branch Testing: Branch testing seeks to ensure that every branch has been executed.
Testing	Conditional Testing: In conditional testing, each clause in every condition is forced to take on each of its possible values in combination with those of other clauses.
Testing	Expression Testing: Expression testing requires that every expression assume a variety of values during a test in such a way that no expression can be replaced by a simpler expression and still pass the test.
Testing	Path Testing: In path testing, data is selected to ensure that all paths of the program have been executed.

Erroneous Testing

- Techniques that focus on assessing the presence or absence of errors in the programming process are called error-oriented. There are three broad categories of such techniques: statistical assessment, error-based testing, and fault-based testing. These are stated in order of increasing specificity of what is wrong with the program without reference to the number of remaining faults.
- Error-based testing attempts to show the absence of certain errors in the programming process. Fault-based testing attempts to show the absence of certain faults in the code. Since errors in the programming process are reflected as faults in the code, both techniques demonstrate the absence of faults.

Erroneous Testing

Fault Estimation: Fault seeding is a statistical method used to assess the number and characteristics of the faults remaining in a program. Harlan Mills originally proposed this technique and called it error seeding.

Input Testing: The goal of input testing is to discover input faults by ensuring that test data limits the range of undetected faults.

Perturbation Testing: Perturbation testing attempts to decide what constitutes a sufficient set of paths to test. Faults are modeled as a vector space, and characterization theorems describe when sufficient paths have been tested to discover both computation and input errors.

Fault-Based Testing: Fault-based testing aims at demonstrating that certain prescribed faults are not in the code. It functions well in the role of test data evaluation. Test data that does not succeed in discovering the prescribed faults is not considered adequate.

Erroneous Testing

Local Extent, Finite Breadth: Input-output pairs of data are encoded as a comment in a procedure, as a partial specification of the function to be computed by that procedure. The procedure is then executed for each of the input values and checked for the output values.

Global Extent, Finite Breadth: In mutation testing, test data adequacy is judged by demonstrating that interjected faults are caught. A program with interjected faults is called a mutant and is produced by applying a mutation operator. Such an operator changes a single expression in the program to another expression, selected from a finite class of expressions.

Local Extent, Infinite Breadth: Rules for recognizing error-sensitive data are described for each primitive language construct. Satisfaction of a rule for a given construct during testing means that all alternate forms of that construct have been distinguished.

Erroneous Testing

- Global Extent, Infinite Breadth: We can define a fault-based method based on symbolic execution that permits elimination of infinitely many faults through evidence of global failures. Symbolic faults are inserted into the code, which is then executed on real or symbolic data..

Stress Testing

The types of internal limitations that can be evaluated with volume testing include:

- Internal accumulation of information, such as tables.
- Number of line items in an event, such as the number of items that can be included within an order.
- Size of accumulation fields.
- Data-related limitations, such as leap year, decade change, switching calendar years, etc.
- Field size limitations, such as number of characters allocated for people's names.
- Number of accounting entities, such as number of business locations, state/country in which business is performed, etc.

Defect Severity

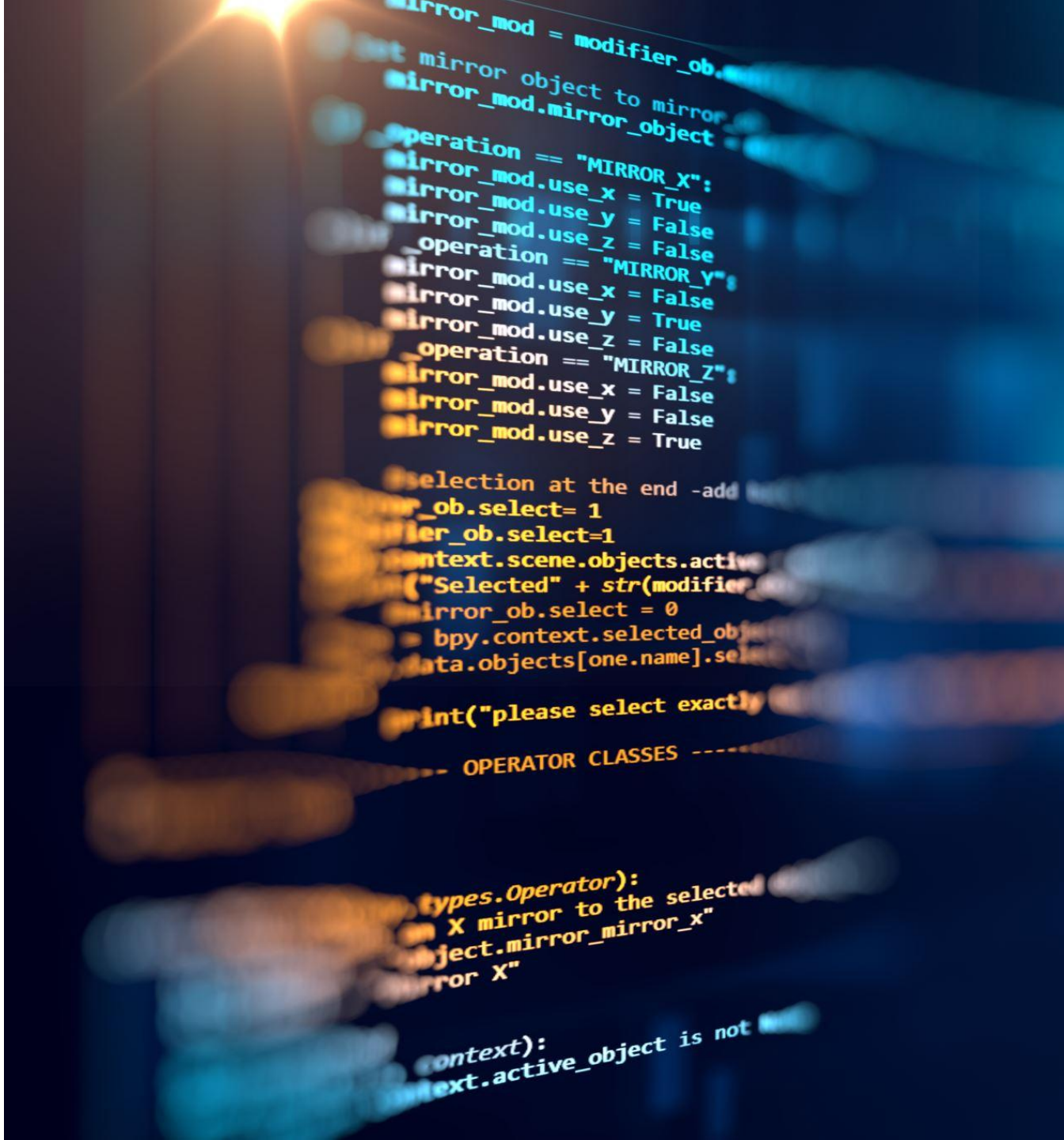
Critical - The defect(s) would stop the software system from operating.

Major - The defect(s) would cause incorrect output to be produced.

Minor - The defect(s) would be a problem but would not cause improper output to be produced, such as a system documentation error.

Defect Classes

- • **Missing** - A specification was not included in the software.
- • **Wrong** - A specification was improperly implemented in the software.
- • **Extra** - An element in the software was not requested by a specification



Defect Naming

- • **Name** – Requirement defect
- • **Severity** – Minor
- • **Type** - Procedural
- • **Class** – Missing

Test Reports

- Current Status Test Reports
- Defect Status Report
- Functional Testing Status Report
- Expected versus Actual Defects Uncovered Timeline
- Defects Uncovered versus Corrected Gap Timeline
- Average Age of Uncorrected Defects by Type
- Defect Distribution Report
- Relative Defect Distribution Report
- Testing Action Report
- Individual Project Component Test Results
- Summary Project Status Report

Risks with New Technology

- Is new technology utilized on the project being tested?
- If so, what are the concerns and risks associated with using that technology?
- If significant risks exist how will the testing process address those risks?

Special Types of Tests

Vendor Validation Testing verifies if the functionalities of the third-party software meet the organization's requirements

Conversion Testing is specifically designed to validate the effectiveness of the conversion process

Stress testing is one which deliberately stresses a system by pushing it to and beyond its specified limits. It is testing the system at and beyond its maximum planned output and determining the point where the system breaks.

Load Testing is where additional load is simulated either manually or using tools to determine how the system will performance in real life situations.

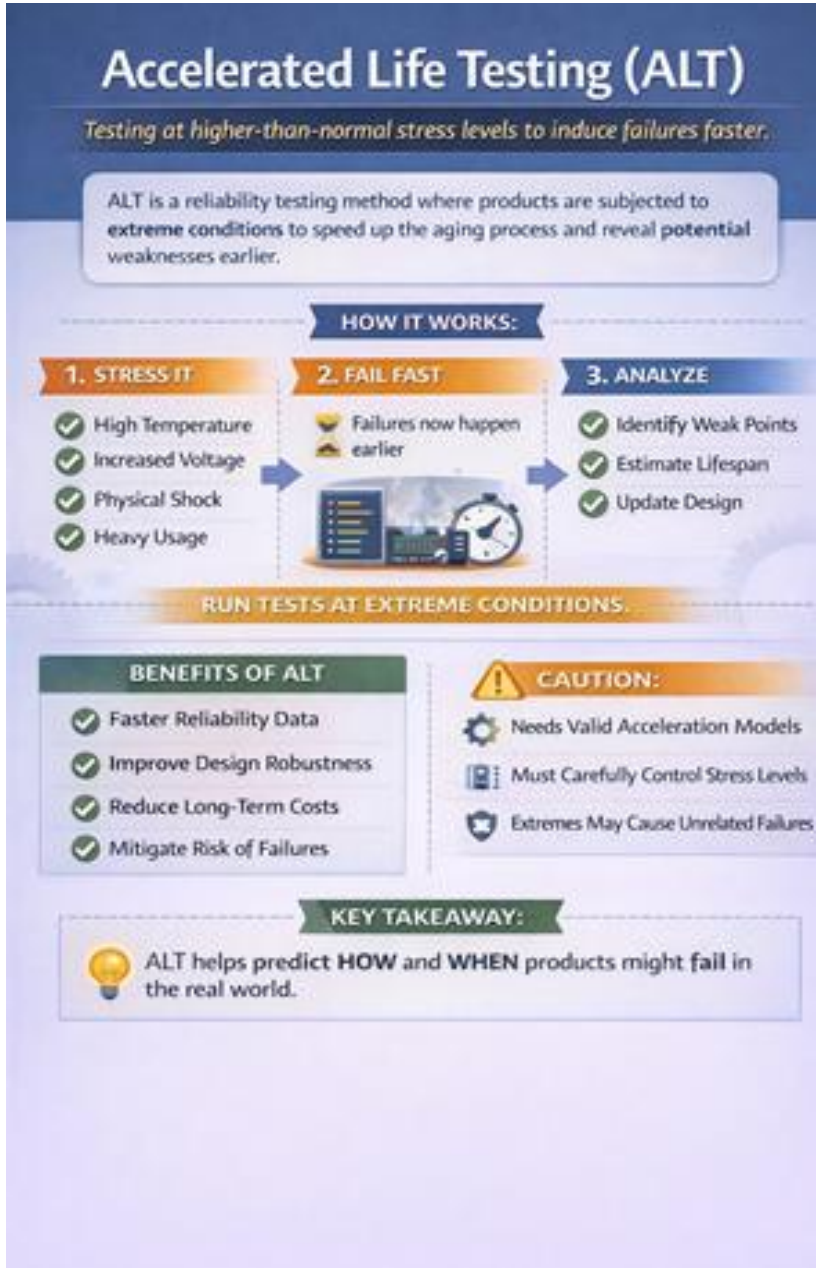
Special Types of Tests

Performance Testing is comparing and measuring the performance of the application against a pre-determined or expected level that is pre-defined by the business users.

Recovery Testing evaluates the contingency features built into the application. Tests the backup, restoration and restart capabilities

Configuration Testing validates if the application can be configured to run in different operating systems and or different versions of internet browsers including Netscape, and modem speeds etc.

Benefit Realization Testing is conducted after the application is moved to production to determine if it delivers the original projected benefits. The analysis is usually conducted by the business users and results are delivered to executive management.

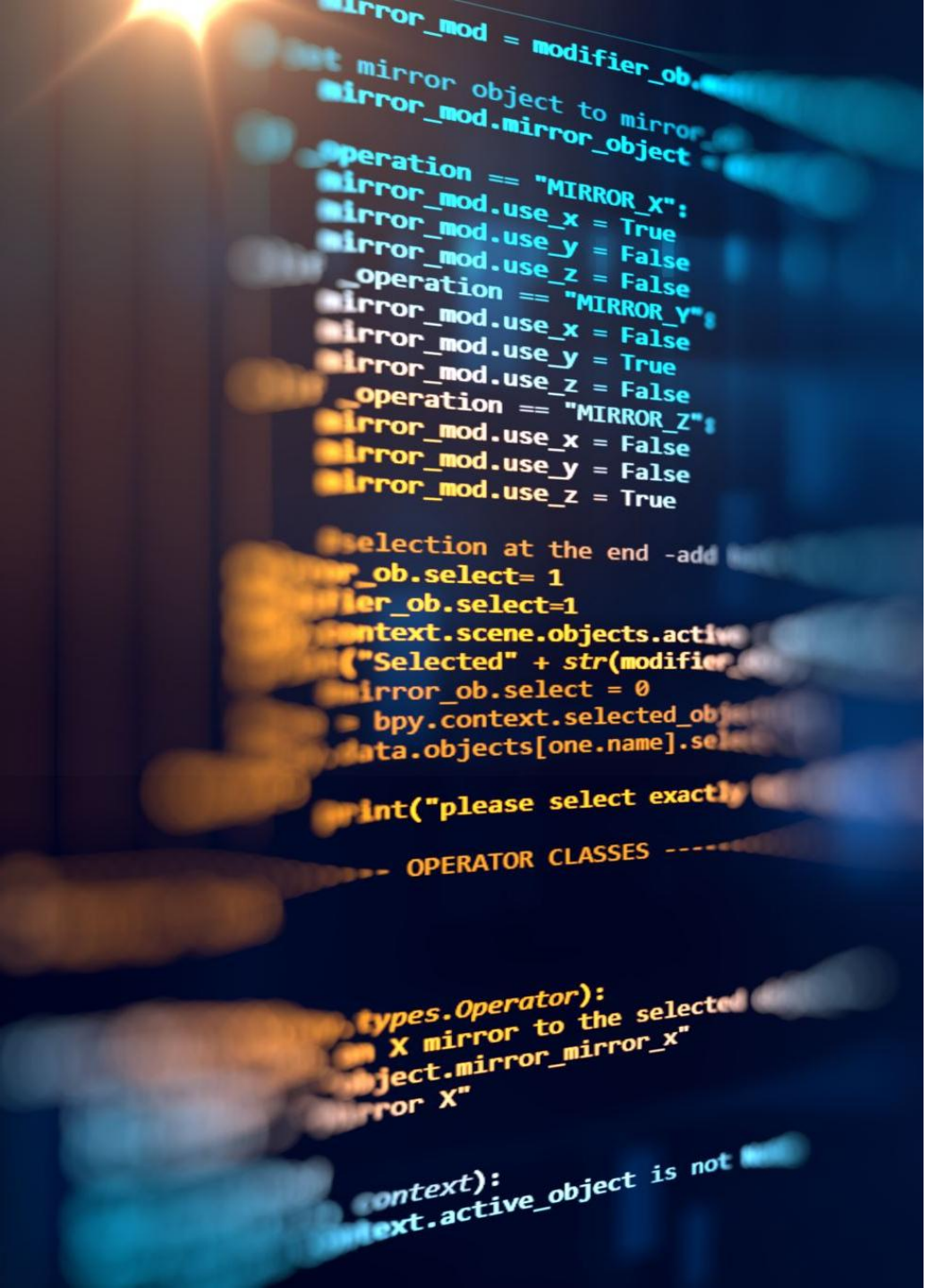


Accelerated Life Testing (ALT)

- Accelerated Life Testing (ALT) is a reliability testing technique in which a product is subjected to higher-than-normal stress conditions – such as increased temperature, voltage, pressure, or usage rates – to force failures to occur more quickly than they would under normal operating conditions. By accelerating the aging or degradation process, engineers can observe failure modes, estimate product lifespan, and identify weaknesses in a much shorter time frame. The results are then extrapolated back to normal conditions using statistical models, allowing organizations to predict long-term reliability, improve design robustness, and reduce the risk of failures in real-world use without waiting for years of actual operation.

Failure Mode and Effects Analysis (FMEA)

Failure Mode and Effects Analysis (FMEA) is a systematic, proactive method used to identify and evaluate potential failures in a product, process, or system before they occur. It involves examining each component or step to determine possible failure modes, assessing the effects of those failures on overall performance, and prioritizing them based on factors such as severity, likelihood of occurrence, and detectability. By calculating a risk priority number (RPN) or similar metric, teams can focus on the most critical risks and implement corrective actions to reduce or eliminate them. FMEA is widely used in engineering, manufacturing, and quality assurance to improve reliability, enhance safety, and prevent costly defects or system breakdowns.

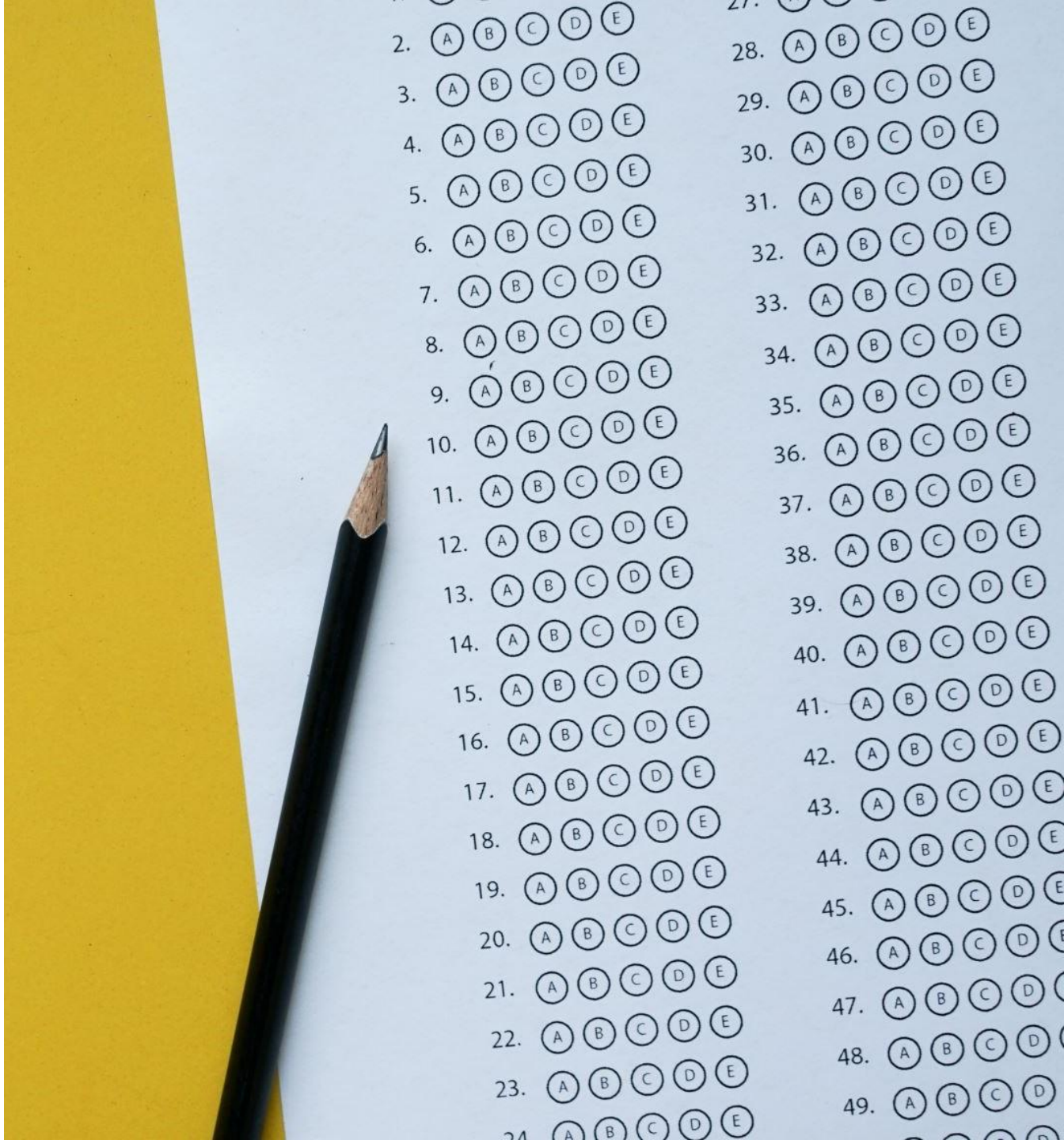


Scripts

- *Unit Scripting* – Develop a script to test a specific unit or module.
- *Pseudo-concurrency Scripting* – Develop scripts to test when there are two or more
- users accessing the same file at the same time.
- *Integration Scripting* – Determine that various modules can be properly linked.
- *Regression Scripting* – Determine that the unchanged portions of systems remain unchanged when the system is changed. *Stress and Performance Scripting* – Determine whether the system will perform
- correctly when it is stressed to its capacity. This

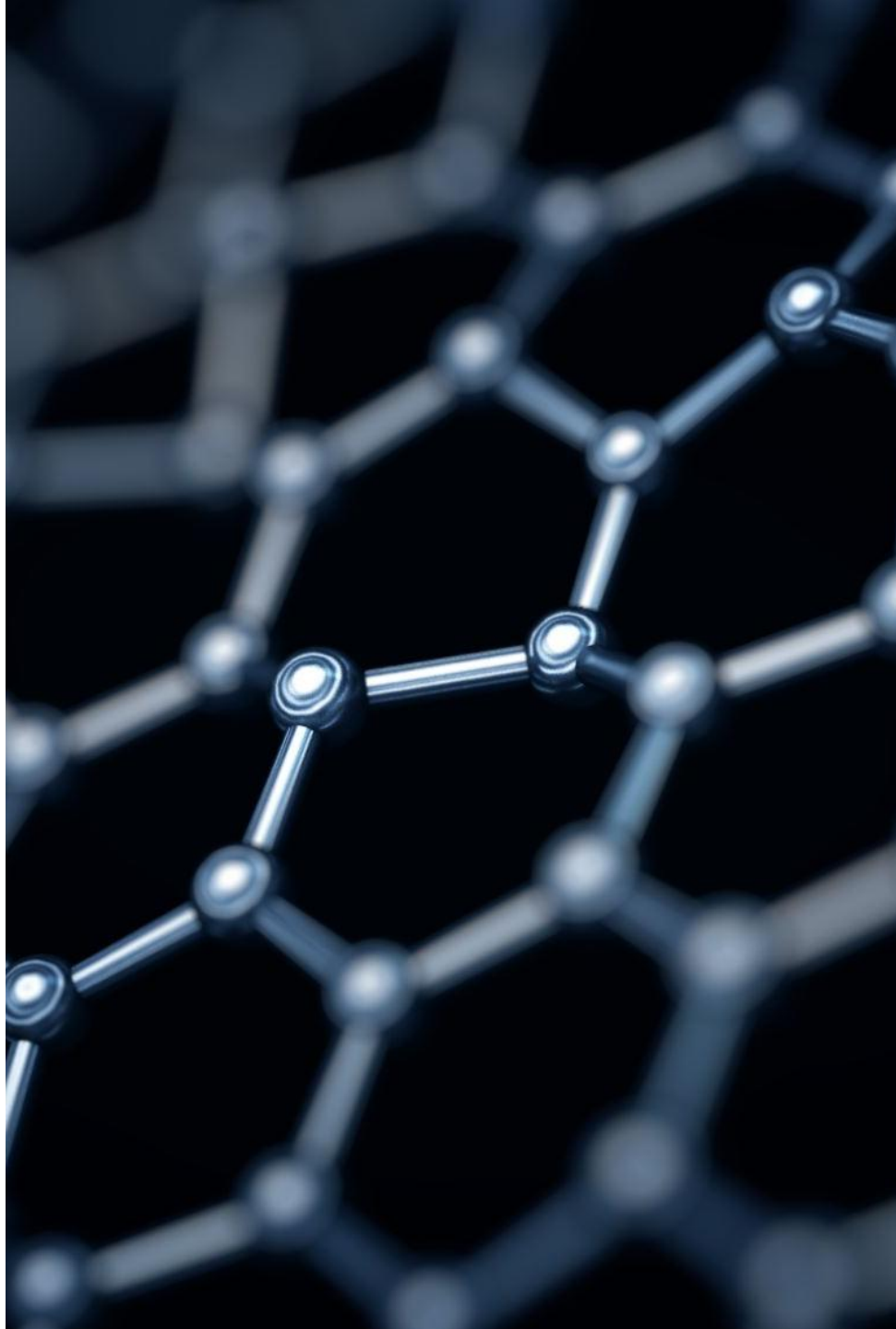
Items for Scripts

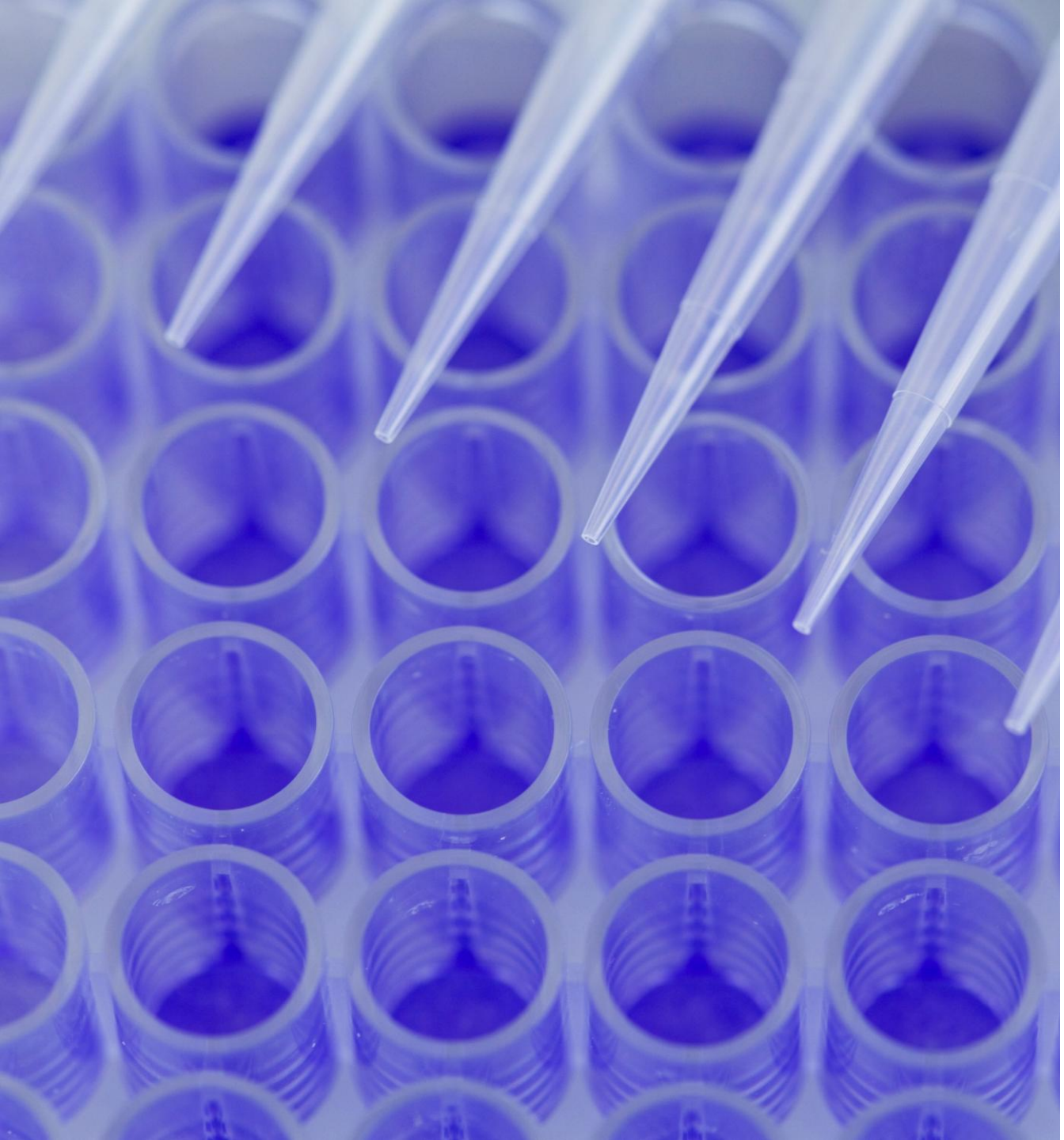
- • Test Item – a unique item identified of the test condition.
- • Entered by – Who will enter the script.
- • Sequence – The sequence in which the actions are to be entered.
- • Action – The action or scripted item to be entered.
- • Expected Result – The result expected from entering the action.
- • Operator Instructions – What the operator is to do if the proper result is received, or if an improper result is returned.



Process for building test cases

Identify test resources.
Identify conditions to be tested.
Rank test conditions.
Select conditions for testing.
Determine correct results of processing.
Create test cases.
Document test conditions.
Conduct test.
Verify and correct.





Test Coverage

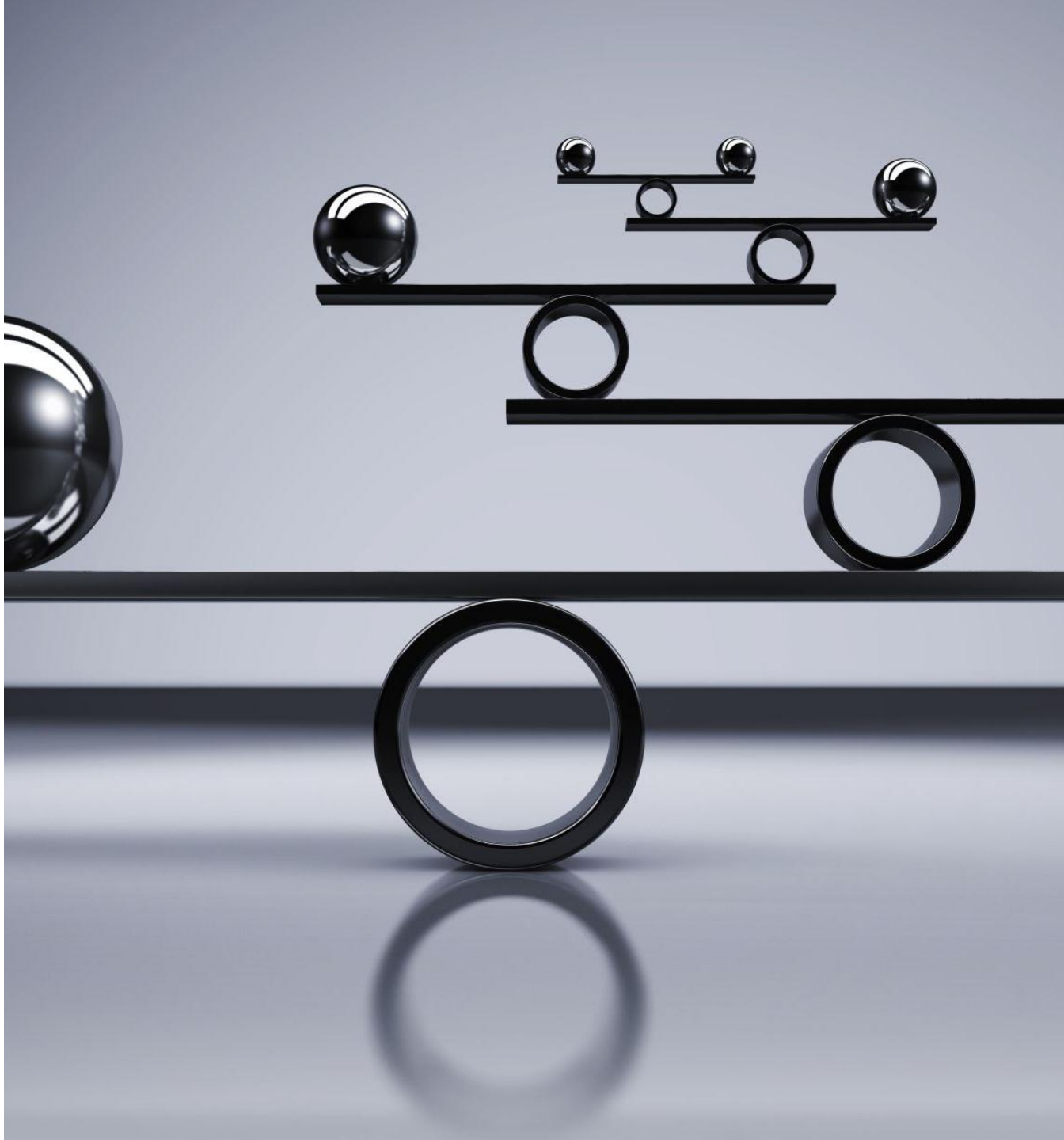
- Based upon the risk, and criticality associated with the application under test, the project team should establish a coverage goal during test planning. The coverage goal defines the amount of code that must be executed by the tests for the application.

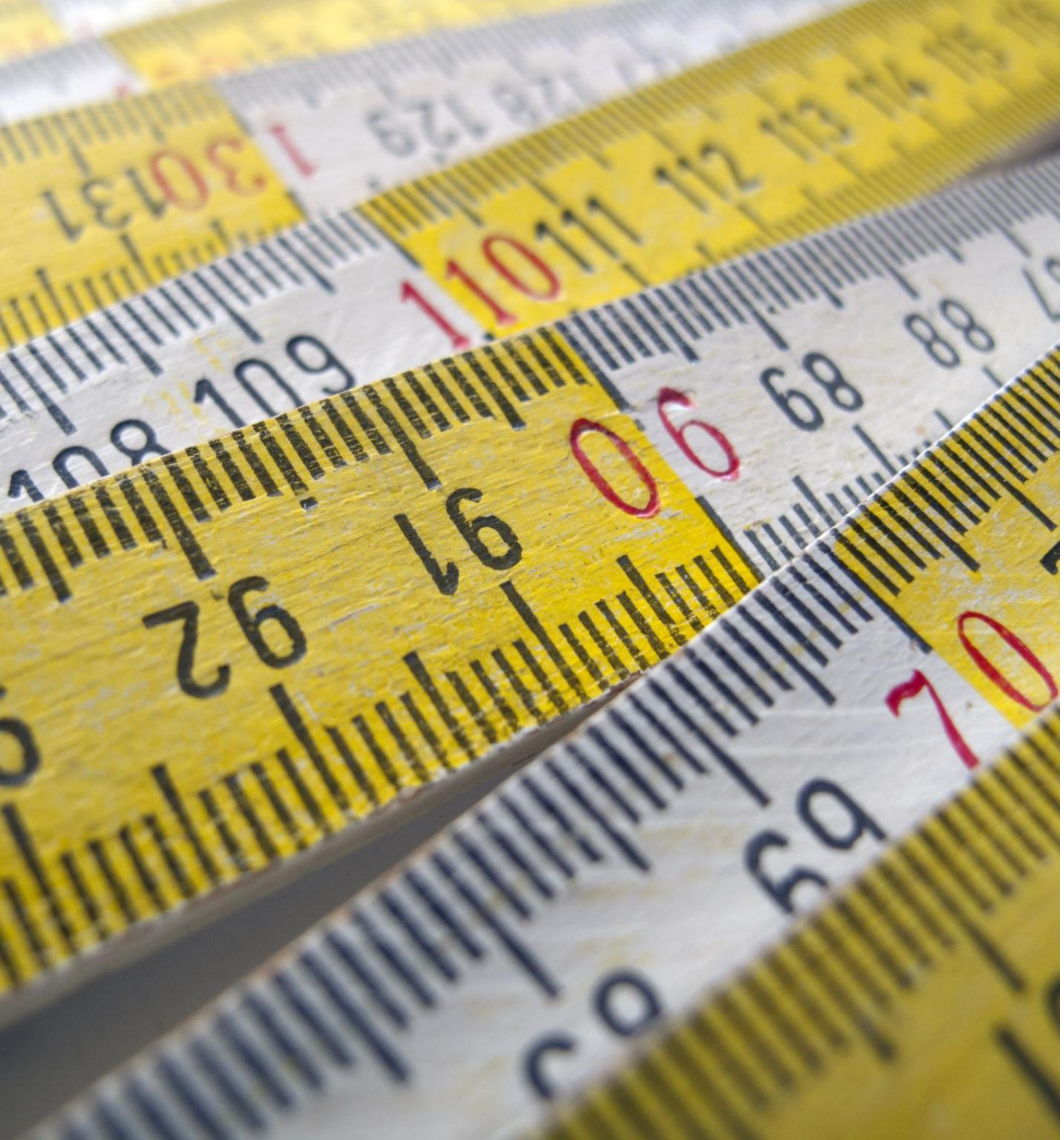
Performing Tests

- Test platforms
- Test cycle strategy
- Use of tools in testing
- Test execution
- Executing the Unit Test plan
- Executing the Integration Test Plan
- Executing the System Test Plan
- When is Testing Complete?
- Concerns

Good Metrics

- Reliability
- Validity
- Ease of Use and Simplicity
- Timeliness
- Calibration





Test Metric Categories

- Metrics unique to test
- Complexity measurements
- Project metrics
- Size measurements
- Defect metrics
- Product measures
- Satisfaction metrics
- Productivity metrics

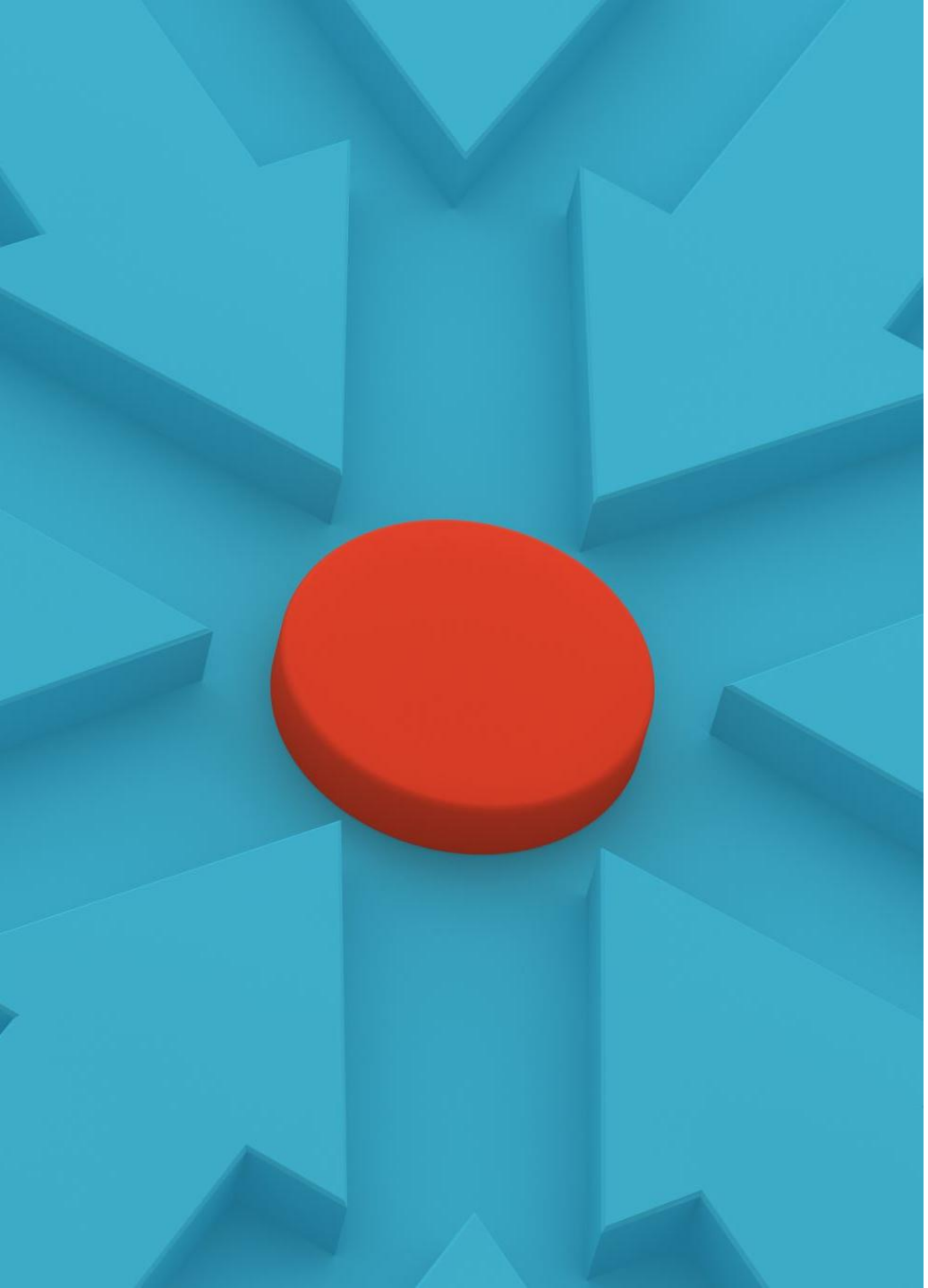
Acceptance Test Planning

- • Acceptance Criteria
- • Acceptance Test Plan
- • Use Case Test Data

Example of Acceptance Criteria

Table from CBK

Criteria	Action
Hardware/Software Project	The name of the project being acceptance-tested. This is the name the user or customer calls the project.
Number	A sequential number identifying acceptance criteria.
Acceptance Requirement	A user requirement that will be used to determine whether the corrected hardware/software is acceptable.
Critical / Non -Critical	Indicate whether the acceptance requirement is critical, meaning that it must be met, or non-critical, meaning that it is desirable but not essential.
Test Result	Indicates after acceptance testing whether the requirement is acceptable or not acceptable, meaning that the project is rejected because it does not meet the requirement.
Comments	Clarify the criticality of the requirement; or indicate the meaning of the test result rejection. For example: The software cannot be run; or management will make a judgment after acceptance testing as to whether the project can be run.



Objectives of SAST

- SAST aims to detect application security vulnerabilities and their root causes when code is not running. Static Application Security Testing (SAST) is a type of security testing methodology that analyzes source code, bytecode, or binary code for vulnerabilities without executing the program. It is typically performed during the early phases of software development, particularly during coding or before deployment, to identify potential security issues that could be exploited by attackers.
- White-box testing: SAST has full knowledge of the application's internal structure. It examines the source code, configuration files, and other design elements.
- Early detection: It helps developers catch vulnerabilities early in the Software Development Life Cycle (SDLC), which is typically more cost-effective than fixing them after deployment.
- Language-specific: SAST tools are usually tailored for specific programming languages or platforms (e.g., Java, C#, Python).
- Automated analysis: Modern SAST tools automate scanning and provide detailed reports of potential vulnerabilities.

Why SAST

- Many serious security vulnerabilities can be addressed
- Decreases the remediation cost of vulnerabilities in later stages
- Deeper analysis is possible with less false positive and negative
- Help you to add best secure coding practices as per department coding standards which can prevent future security issues

Defects Static Analysis can Catch

Defects that result from inconsistently following simple, mechanical design rules.

- **Security:** Buffer overruns, improperly validated input.
- **Memory safety:** Null dereference, uninitialized data.
- **Resource leaks:** Memory, OS resources.
- **API Protocols:** Device drivers; real time libraries; GUI frameworks.
- **Exceptions:** Arithmetic/library/user-defined
- **Encapsulation:** Accessing internal data, calling private functions.
- **Data races:** Two threads access the same data without synchronization

Skills Required for SAST

Working knowledge of the programming language under review

Knowledge of standard, secure coding techniques

Quickly able to detect insecure coding practices

Analytical skills

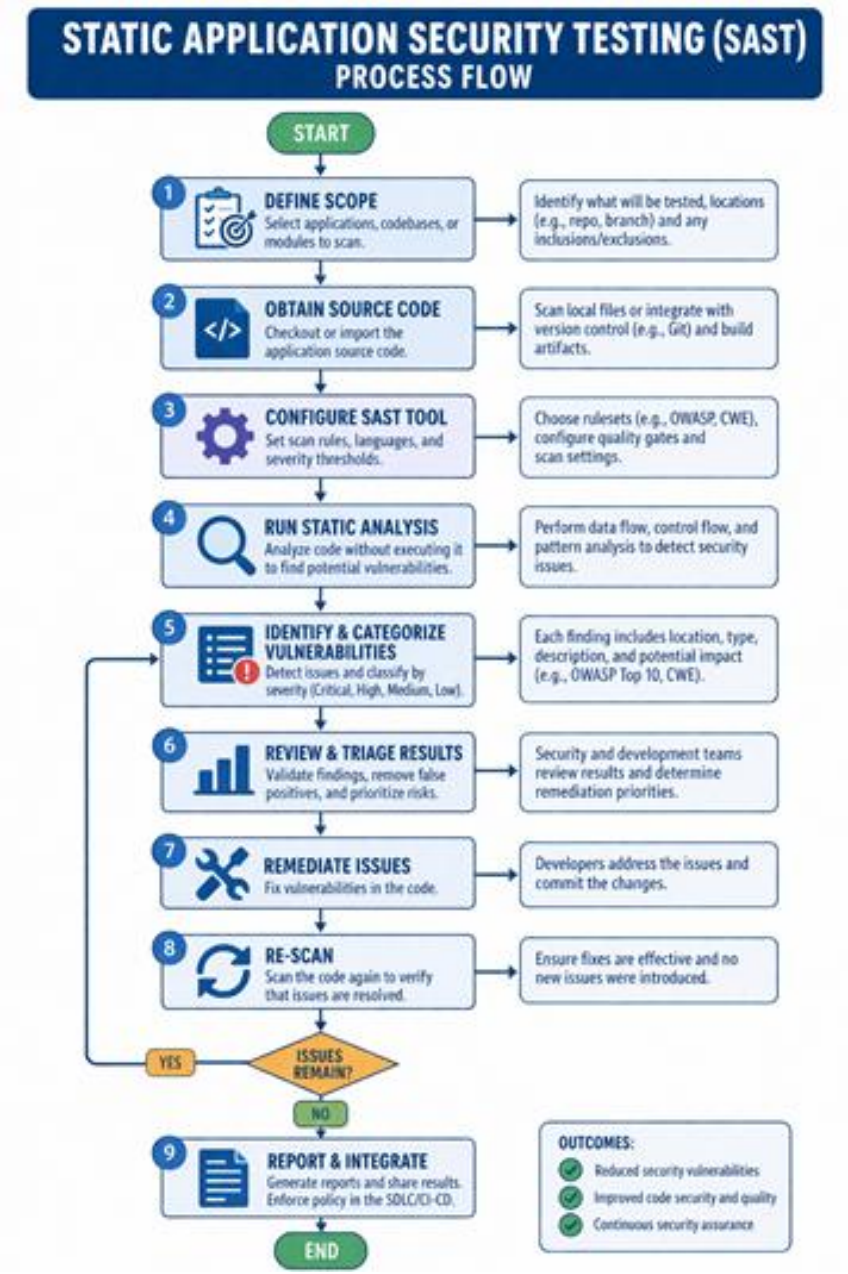
What to Look for in SAST

- Look for the code that implements common application security mechanisms
- The insecure implementation of security mechanisms can render applications vulnerable to various attacks
- Authentication
- Session Management
- Error Handling
- Encryption
- Authorization
- Data Validation
- Logging
- Security Configuration

SAST Steps

- **Perform an application walkthrough**
  - Understand the applications functionality
- **Developers Interview**
  - Talk to developers; more you involve developers, more effective you secure code review will be
  - It will help you quickly
  - Determine the high risk areas
  - Understand developer's logic behind particular functionality
  - Establish friendly and respectful relationship between you and developers
- **Analyze threat Model**
  - Figure out the major existing security threats
- **Define secure code review objectives**
  - Based on threat model analysis, identify the objectives such as:
  - Code review for all major security vulnerabilities
  - Code review for OWASP top 10
  - Code review for specific vulnerability
- **Define scope for secure code review**
  - Be clear on what to look for and what to avoid
  - Calculate the time and efforts required
  - Define the budget constraints
- **Categorized the Objectives**
  - Prioritize the objective based on severity and requires more attention
- **Conduct Static Code Analysis**
  - Use multiple source code analysis tools to perform secure source code analysis
  - Most common vulnerabilities and flaws are identified
- **Conduct Manual Source code Review**
  - Inspect the application's code line by line in order to find the use of insecure coding practices
- **Remediation/Recommendation**
  - Suggest secure coding practice for each findings
- **Prepare Report**

SAST Activities-flow Chart



SAST Deliverable

- **SAST report should include:**

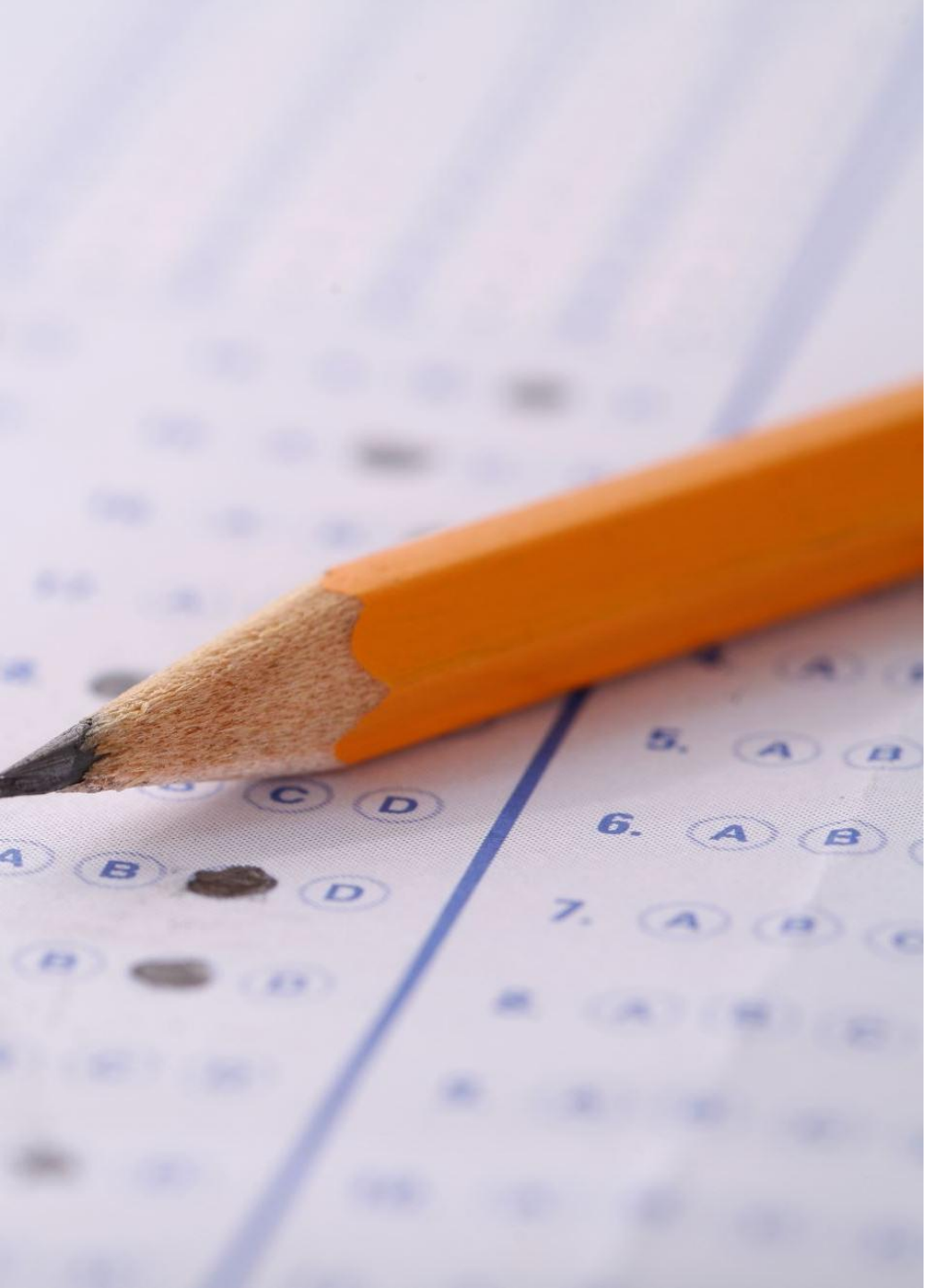
- Types of vulnerabilities
- Severity level/Impact
- Location, line of code where security issue exists
- Recommendations

SAST vs DAST

SAST	DAST
White box security testing	Black box security testing
Requires a source code	Requires a running application
Finds vulnerability earlier in SDLC	Finds the vulnerability towards the end of SDLC
Less expensive to fix vulnerability	More expensive to fix vulnerability
Runtime and environment related issues can’t be discovered	Runtime and environment related issues can be discovered
Typically supports all kinds of software	Typically scans only apps like web application and web services

SCA

Third-party dependencies are one of the biggest concerns when we deal with code. 80% to 90% of the software code contains third-party dependencies or libraries (reference: https://snyk.io/reports/open-source-security). These dependencies come with their own issues and benefits. In this chapter, software composition analysis (SCA) and its uses will be discussed. SCA is like a dedicated quality assurance team that covers security and compliance for your software's ingredients. It ensures you're building your software with the most up-to-date, safest, and compliant components available. This not only ensures a better end product but also saves you from potential headaches, be they security breaches or legal battles, down the road.
-Sehgal, Vandana Verma. Implementing DevSecOps Practices: Understand application security testing and secure coding by integrating SAST and DAST (pp. 201-202). Packt Publishing.



What is Test Driven Development

- Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. It's a practice that's often associated with Agile development methodologies. The basic idea is to write and correct the failed tests before writing new code (the opposite of traditional testing methods where code is developed first and tested after). This approach emphasizes a short development cycle to increase productivity and ensure that the code meets the requirements from the beginning.

History of TDD

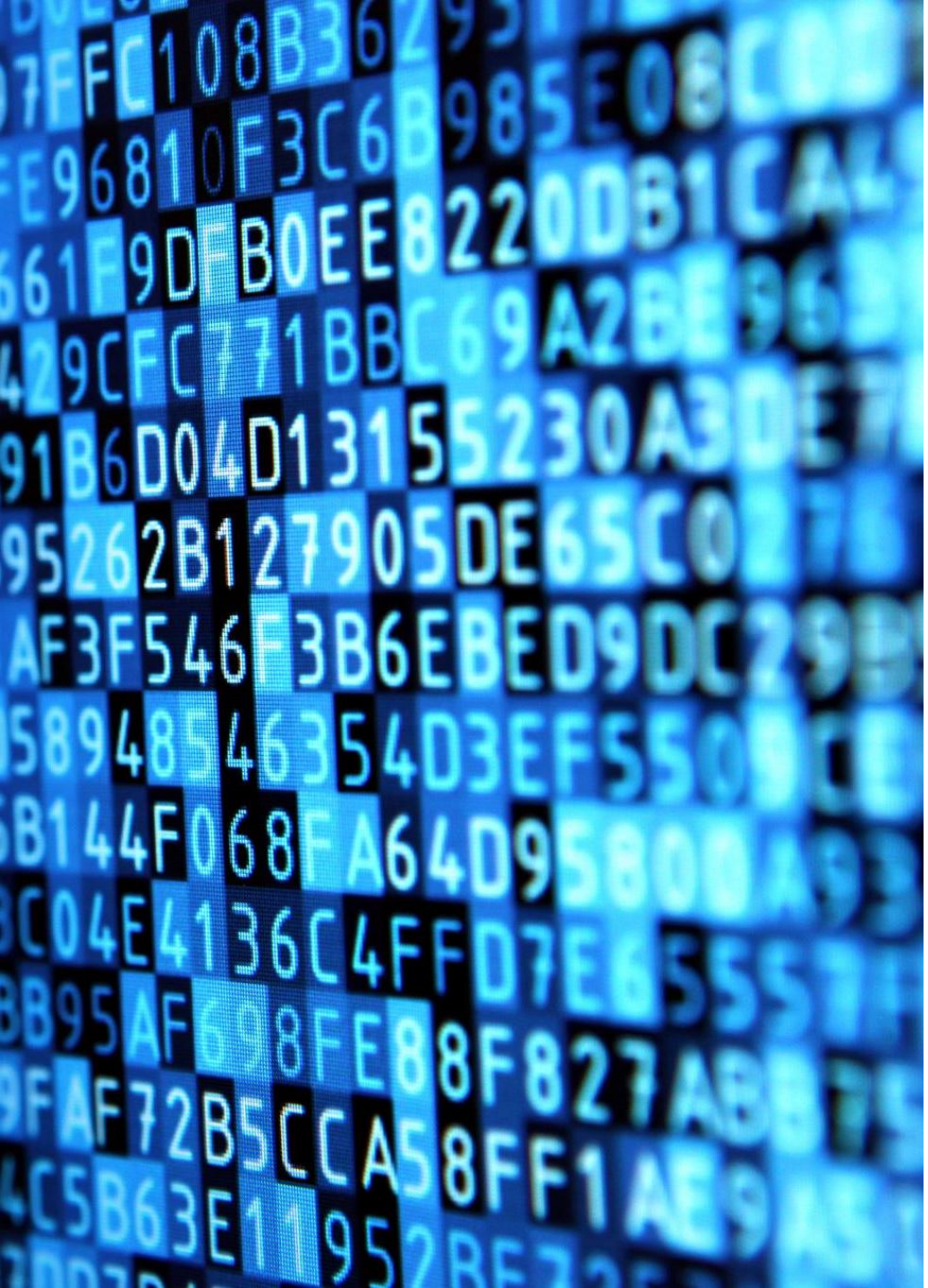
The history of TDD starts in 1999 with a group of developers who championed a set of concepts known as Extreme Programing (XP). XP is an agile based methodology that is based on recognizing what practices in software development are beneficial and dedicating the bulk of the developers time and effort to those practices under the philosophy “if some is good, more is better.” A key component of XP is test-first programming. TDD grew out of XP as some developers found they were not ready to embrace some of the more, at the time, radical concepts, yet found the promise of improved quality that was delivered by the practice of TDD compelling.

-Bender, James; McWherter, Jeff. Professional Test Driven Development with C#: Developing Real World Applications with TDD (p. 16). Wiley. Kindle Edition.

TDD

Kent Beck, in his book Test-Driven Development: By Example (Addison-Wesley Professional, 2003), defines test-driven development using the following rules: Never write a single line of code unless you have a failing automated test. Eliminate duplication.

-Vorontsov, Alexei; Newkirk, James W.. Test-Driven Development in Microsoft .NET (Developer Reference) . Pearson Education.



TDD

- So, you should always strive to do no more and no less, as follows: The code is appropriate for the intended audience. The code passes all the tests. The code communicates everything it needs to. The code has the smallest number of classes. The code has the smallest number of methods.

- Vorontsov, Alexei; Newkirk, James W.. Test-Driven Development in Microsoft .NET (Developer Reference). Pearson Education.

What is Test Driven Development

- With Test-Driven Development (TDD) software requirements are converted to test cases before the software is developed. Development is tracked by repeatedly testing the software against the test cases.

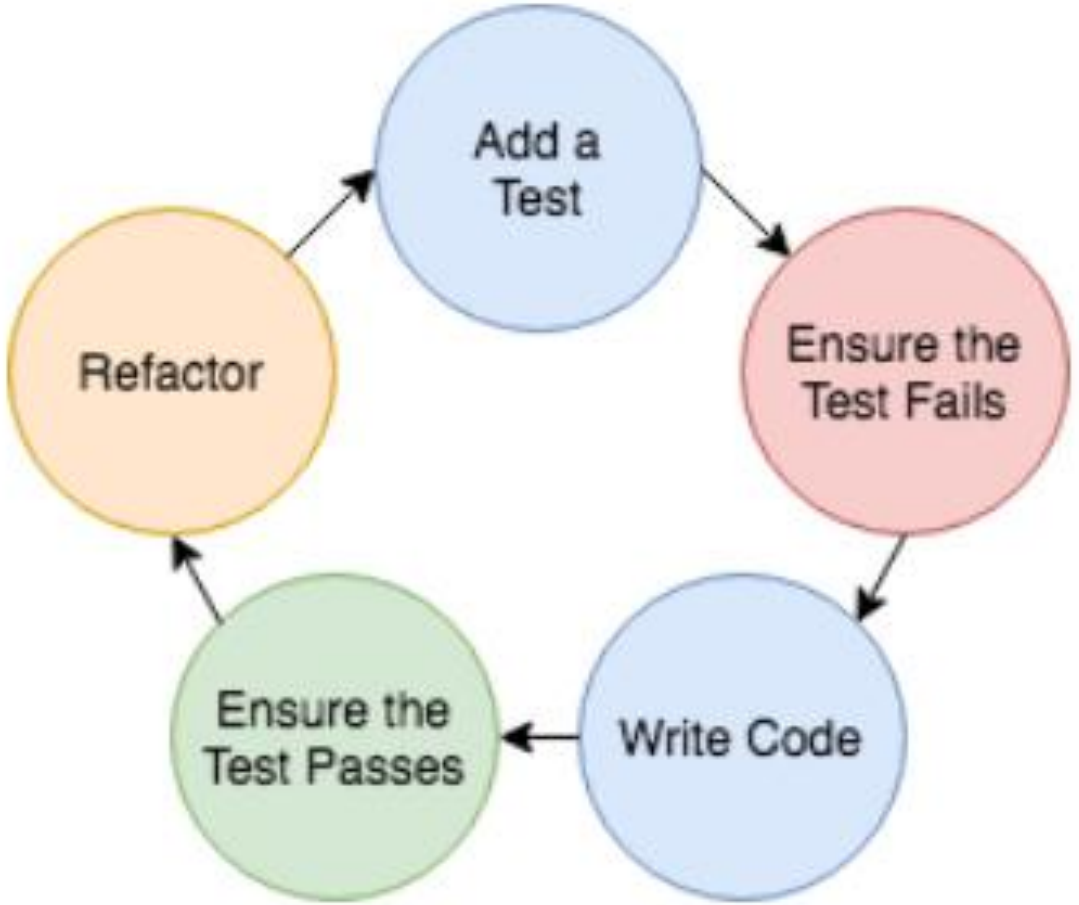
- There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS). By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods.

What is Test Driven Development

- The tests should be written before the functionality that is to be tested. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later. It also ensures that tests for every feature gets written.
- Each test case fails initially: This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has led to the "test-driven development mantra", which is "red/green/refactor", where red means *fail* and green means *pass*

Red-Green - Refactor

- Add a test to the test suite
- (**Red**) Run all the tests to ensure the new test fails
- (**Green**) Write just enough code to get that single test to pass
- Run all tests
- (**Refactor**) Improve the initial code while keeping the tests green
- Repeat



Refactoring

Refactoring is defined as improving the code while not changing its functionality. The definitive book on refactoring is Martin Fowler's, Refactoring: Improving the Design of Existing Code (Addison-Wesley, 1999).

Refactoring is a critical part of TDD because you need to refine the code's design as you add additional tests. For example, if you see duplicated code in the solution, you need to remove it. If you need to introduce complexity to remove the duplication, it is all right because there is an actual need, not an anticipated need. Following this practice diligently yields an implementation that is complex where it needs to be; no more and no less. It is through refactoring and simple design that you can refine the system's design to meet the requirements specified by the tests.

Vorontsov, Alexei; Newkirk, James W.. Test-Driven Development in Microsoft .NET (Developer Reference) (p. 28). Pearson Education. Kindle Edition.

Approach 1: Inside Out

- With the Inside Out (sometimes referred to as the Detroit School of TDD or Classicist) approach, the focus is on the results (or state). Testing begins at the smallest unit level and the architecture emerges organically. Design happens at the refactor stage, which can unfortunately result in large refactoring's.

Approach 2: Outside-In

- The Outside In (sometimes referred to as the London School of TDD or Mockist) approach focuses on user behavior. Testing begins at the outer-most level and the details emerge as you work your way in. This approach relies heavily on mockups and stubs. Design happens at the red stage.

TDD vs Traditional Testing

- Approach: TDD is an agile development methodology where tests are written before the code is developed
- Testing Scope: TDD focuses on testing small code units at a time
- TDD follows an iterative process, where small chunks of code are developed, tested, and refined until they pass all tests.
- Debugging: TDD aims to catch errors as early as possible in the development process, making debugging and fixing them easier
- Documentation: TDD documentation typically focuses on the test cases and their results, while traditional testing documentation may include more detailed information about the testing process, the test environment, and the system under test

Test Driven Development - Overview

Add a Test: The development process begins by writing a test for a new function or improvement. This test will initially fail because the feature hasn't been implemented yet. The test is designed to define a function or improvements of an existing function.

Run All Tests: After adding the new test, you run all the tests to ensure that the new one fails. This is an important step because it validates that the new test is working correctly and needs new code to pass.

Write the Code: Now, you write the minimum amount of code required to pass the test. This code doesn't have to be perfect; it just needs to work.

Run Tests: Run all the tests again. If all tests pass, you can be confident that the new code meets the test requirements and doesn't break or degrade any existing features.

Refactor Code: Once the test passes, you can clean up your code. This might involve removing duplication, improving performance, or making it more readable. The key here is to make changes without altering the functionality.

Repeat: This process is repeated for each new piece of functionality. Over time, this leads to a comprehensive suite of tests that can be used to ensure that future changes don't break existing functionality.

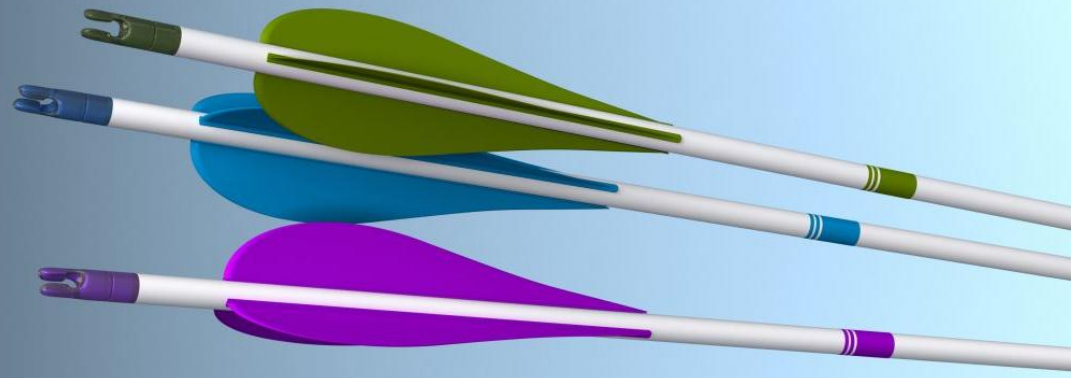
Test Driven Development - Benefits

Early bug detection: Writing tests before code means you're more likely to catch bugs early in the development process.

Better Designed, Cleaner, and More Extendable Code: It helps to write only the code necessary and encourages cleaner, simpler designs.

Confidence to Refactor: With a comprehensive test suite, developers can refactor code with confidence that they haven't broken anything.

Documentation: The tests themselves serve as documentation for the codebase, showing how the system is supposed to behave

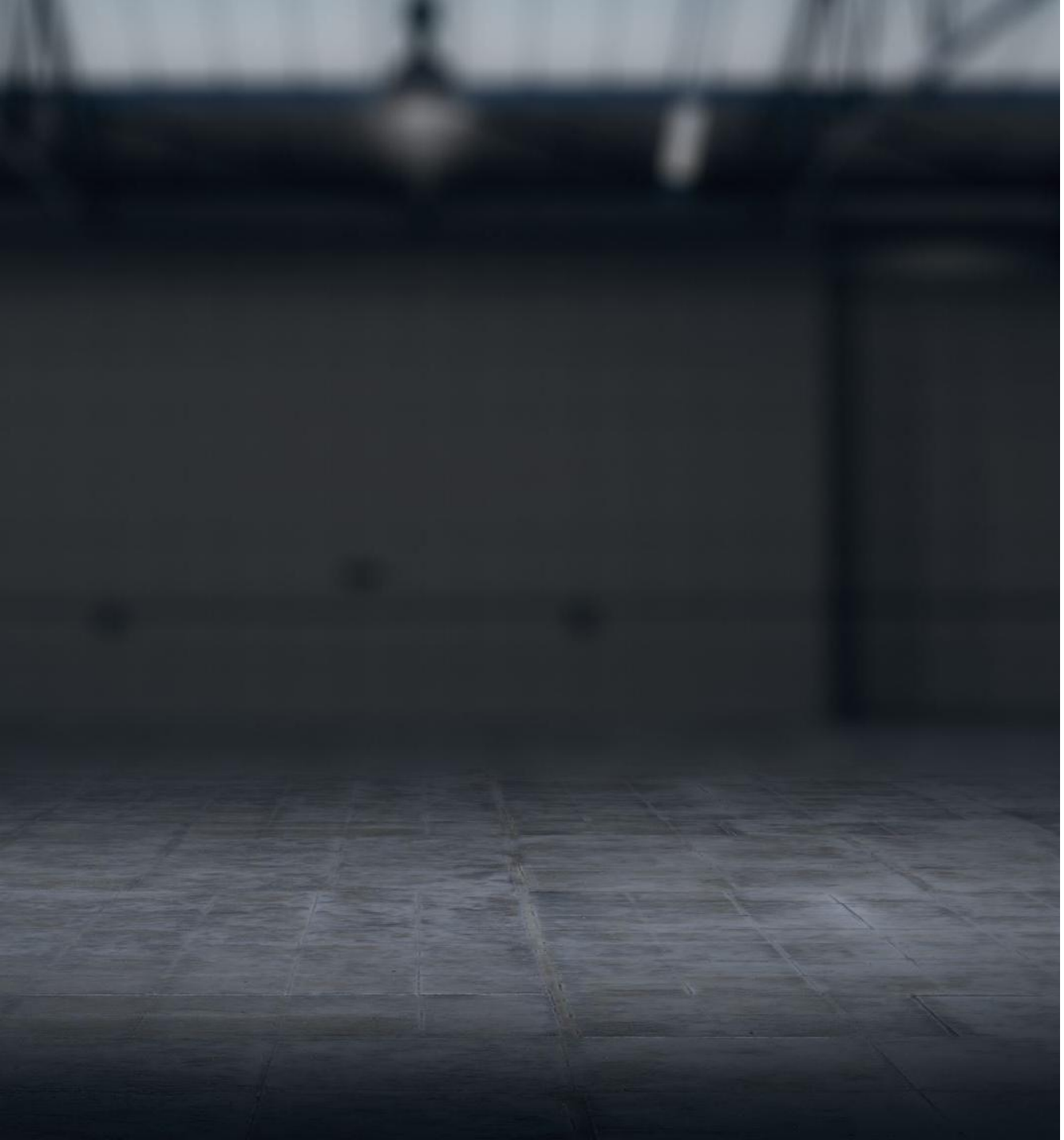


Test Driven Development – Structure

- **Setup**: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- **Execution**: Trigger the UUT to perform the target behavior and capture all output, such as return values and output parameters. Essentially, run a test.
- **Validation**: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT.
- **Cleanup**: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

Test Driven Development – Things to Avoid

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
- Dependencies between test cases. A test suite where test cases are dependent upon each other is complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
- Slow running tests.



SOLID

- **S** - Single-responsibility Principle: A class should have only one job
- **O** - Open-closed Principle: Objects or entities should be open for extension but closed for modification.
- **L** - Liskov Substitution Principle: This means that every subclass or derived class should be substitutable for their base or parent class.
- **I** - Interface Segregation Principle: A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.
- **D** - Dependency Inversion Principle: Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

MSD

- The Mean Squared Deviation formula is relatively simple and provides insight into how any system deviates from expectations (Engel, 2010). This is sometimes referred to as the mean squared error (Engle, 2010; Wasson, 2015). It essentially takes the square of the errors, or deviations from expected/desired outcomes. This formula is shown in 1.

$$\text{MSD} = \frac{1}{n} \sum_{i=2}^{n} (y_i - T)^2$$

- yi is the actual value
- T is the target value.

- This is commonly used in many engineering disciplines. The metric is a positive integer. Interpreting it is quite simple: the closer to zero the MSD is, the more reliable the system in question . It is sometimes the case that one takes the square root of the mean squared error, yielding the root mean squared deviation (or root mean squared error).

MPR

- Mean Percentage Error (MPE) formula. The MPE is the arithmetic mean of errors from modelling (Beynon-Davies, 2016). This metric compares expected values to actual values and calculates mean error. An error is defined as any deviation for the planned or expected value.

$$\mathrm{MPE} = \frac{100\%}{n} \sum_{t=1}^{n} \frac{a_t - f_t}{a_t}$$

- n = is the number of different times for which the variable is forecast.
- a_t is the actual value of the quantity being forecast
- f_t is the forecast.
-
- Essentially this metric is describing the difference between expected values and the actual value

Mean Squared Error

- If a vector of n predictions is generated from a sample of n data points on all variables, and Y is the vector of observed values of the variable being predicted, with $\hat{Y}$ being the predicted values (e.g. as from a least-squares fit), then the within-sample MSE of the predictor is computed as

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Cyclomatic Complexity

Cyclomatic complexity is a software metric used to measure the **complexity of a program's control flow**. It was introduced by Thomas J. McCabe in 1976 and is commonly used in software engineering to evaluate the maintainability, testability, and reliability of a program.

Cyclomatic complexity measures the number of **independent paths** through a program's source code. An independent path is any path through the code that introduces at least one new edge that hasn’t been traversed before in other paths.

The formula to compute it is:

Cyclomatic Complexity (CC)=E−N+2P

Cyclomatic Complexity

- For many functions, you can calculate cyclomatic complexity as:
- CC=Number of decision points+1

- Decision points include:

- if statements

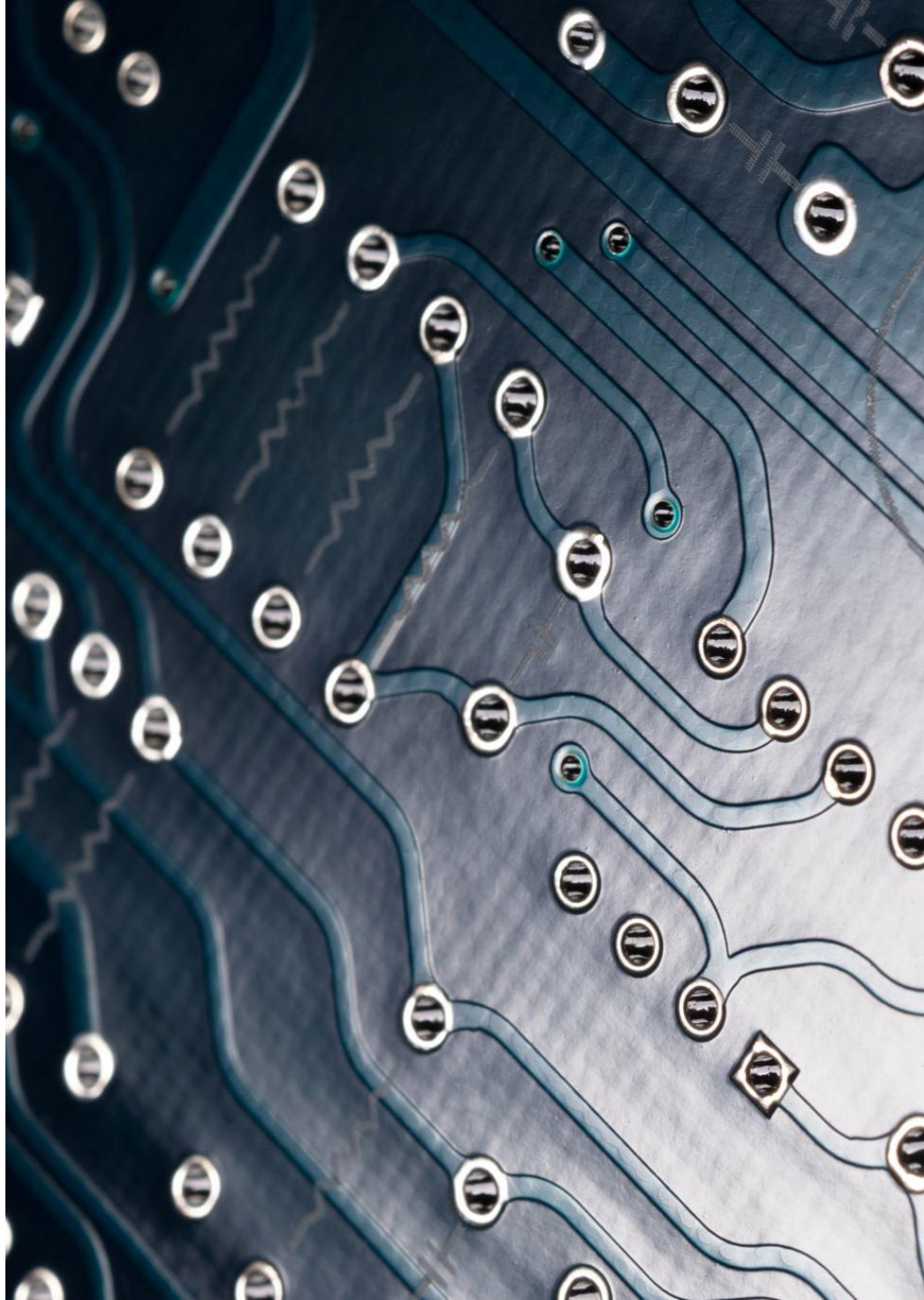
- while, for, and do-while loops

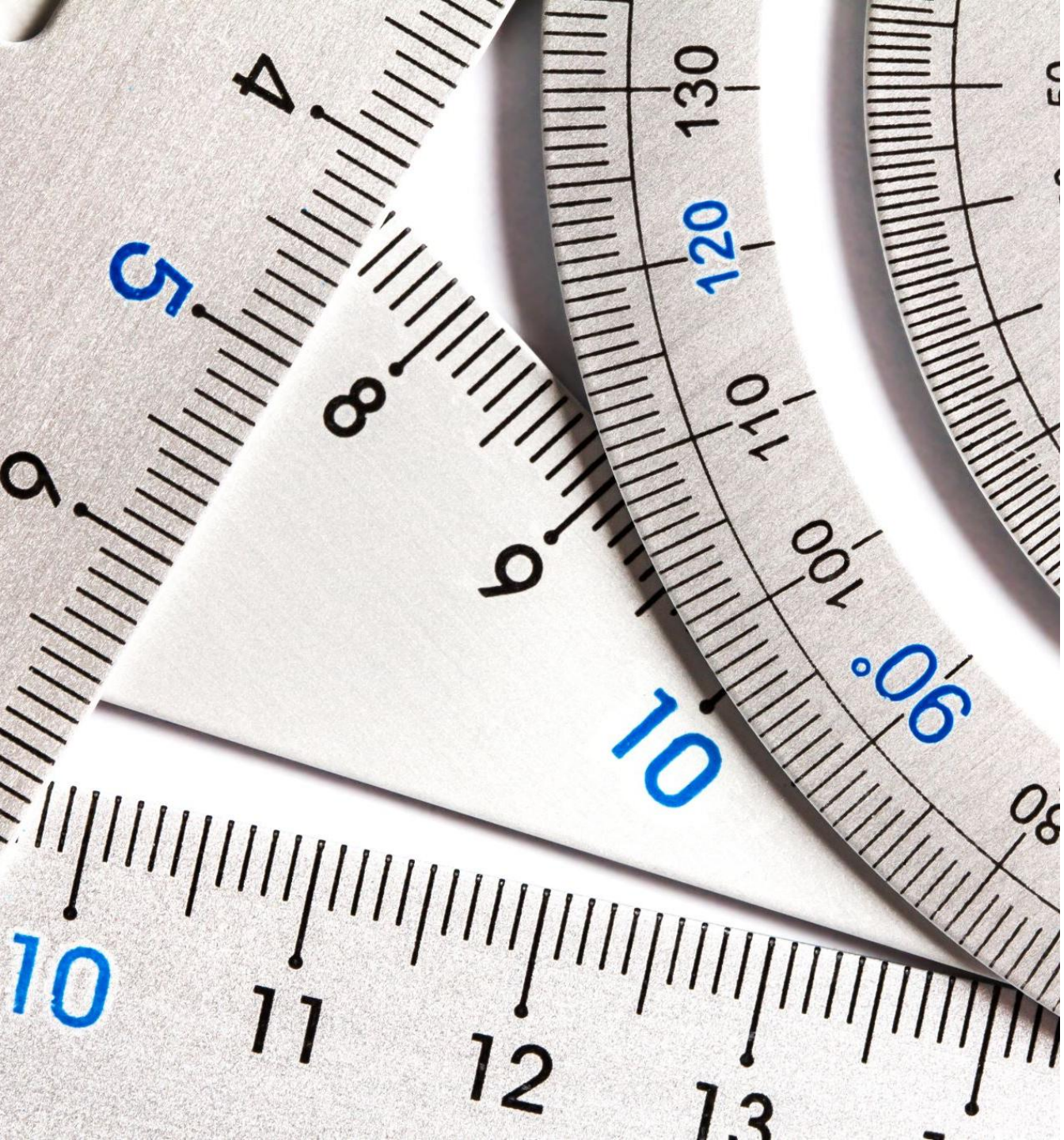
- case statements in a switch

- &&, || in conditionals (each counted as a decision point)

Halstead Metrics

- Halstead Metrics are a suite of **software complexity measures** introduced by Maurice Halstead in 1977. These metrics are based on **counts of operators and operands** in the source code, aiming to quantify the effort and cognitive complexity required to write or understand the software. Halstead metrics treat software like a mathematical formula: a collection of **tokens** (operators and operands). By analyzing these tokens, we can derive various quantitative attributes of the code.
- Key Terminology
- n_1: Number of distinct operators
- n_2: Number of distinct operands
- N_1: Total number of operator occurrences
- N_2: Total number of operand occurrences
- From these, we define:
- $n = n_1 + n_2 \rightarrow$ Total number of unique tokens (vocabulary)
- $N = N_1 + N_2 \rightarrow$ Total number of tokens (length)





Halstead Metrics

- **Program Length (N)**
- N=N1+N2 Total number of operators and operands in the program.

- **Program Vocabulary (n)**
- n=n1+n2

- **Volume (V)**
- V=N×log2(n)
- Represents the size of the implementation in bits. The more complex the vocabulary and length, the higher the volume.

- **Difficulty (D)**

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$$

Measures the difficulty of understanding or writing the code. Higher values imply more challenging code.

- **Effort (E)**
- E=D×V Represents the mental effort needed to write or understand the program.
- **Time to Program (T)**
- T=E/18 Estimated time in seconds to write the program (assuming 18 seconds per mental effort unit).
- **Estimated Bugs**

$$B = \frac{E^{2/3}}{3000}$$

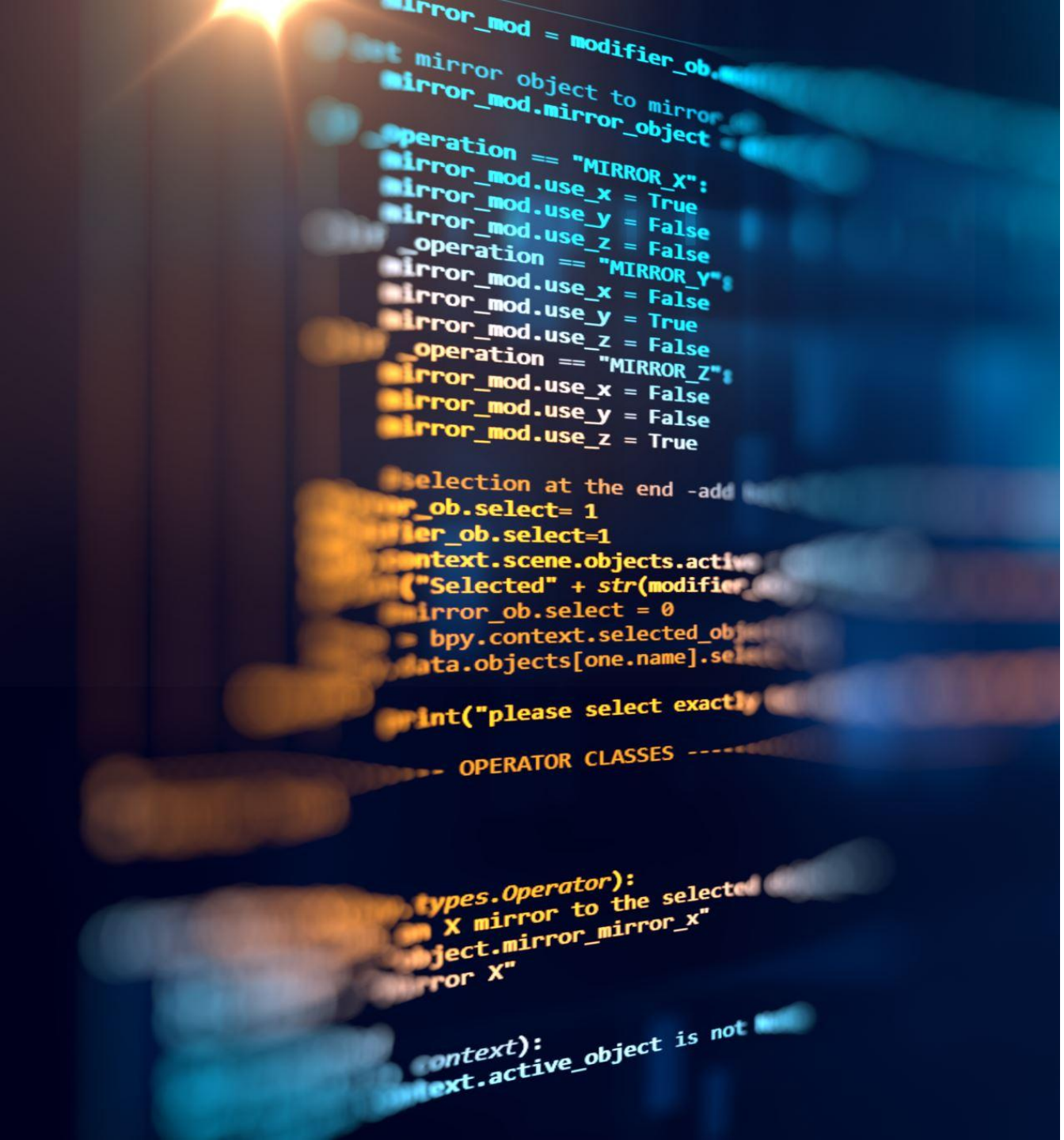
Failure Mode Analysis

Failure Cause: This is what you are seeking, the ultimate cause of the failure. Why did the system fail? This may not always be the proximate cause, or the obvious cause.

Failure Mode: This is a description of what failed. In every engineering discipline a complete description of the failure is critical to understanding why it failed.

Failure Effect: This is the most obvious part of FMEA, what is the effect of the failure. For example, failure of coolant in a Nuclear Reactor could lead to more severe effects.

Failure Severity: This is tightly coordinated

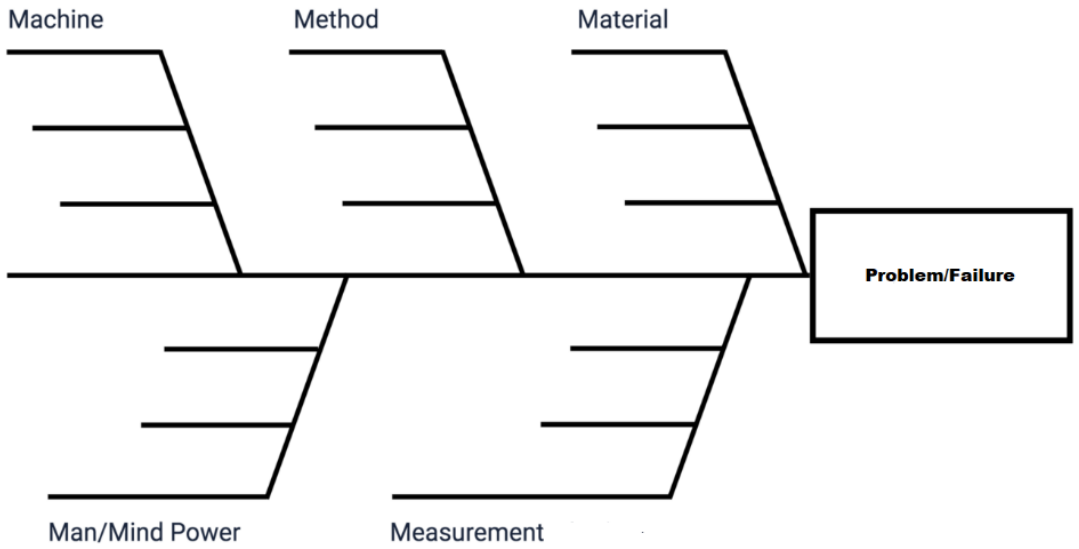


Defect Density

- **Defect Density = (Total Number of Defects)/(Size of the Software)**

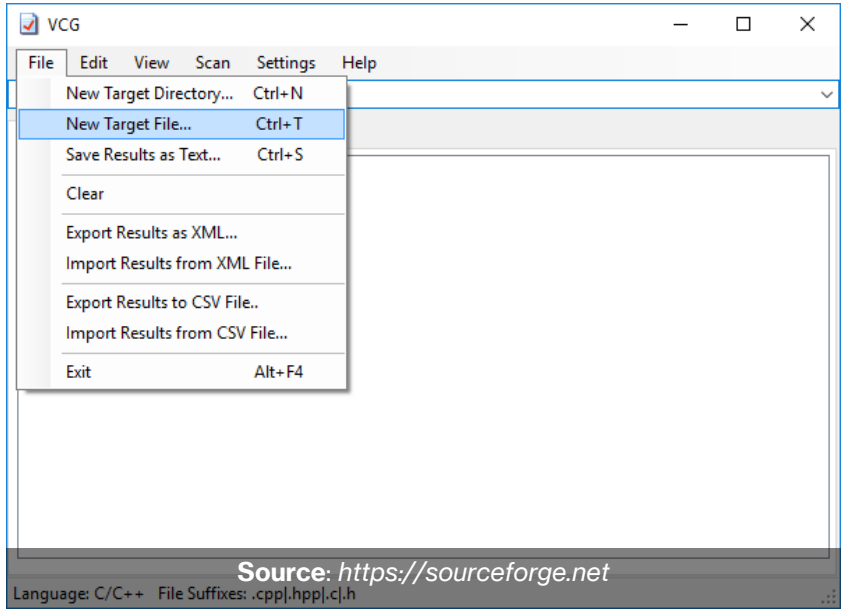
- **Defects can include**
- Can include:
- Logical errors
- Syntax errors (usually caught before production)
- Runtime errors
- UI bugs
- Security vulnerabilities

Failure Mode Analysis – Ishikawa Diagram



Static Code Analysis using Visual Code Grepper (VCG)

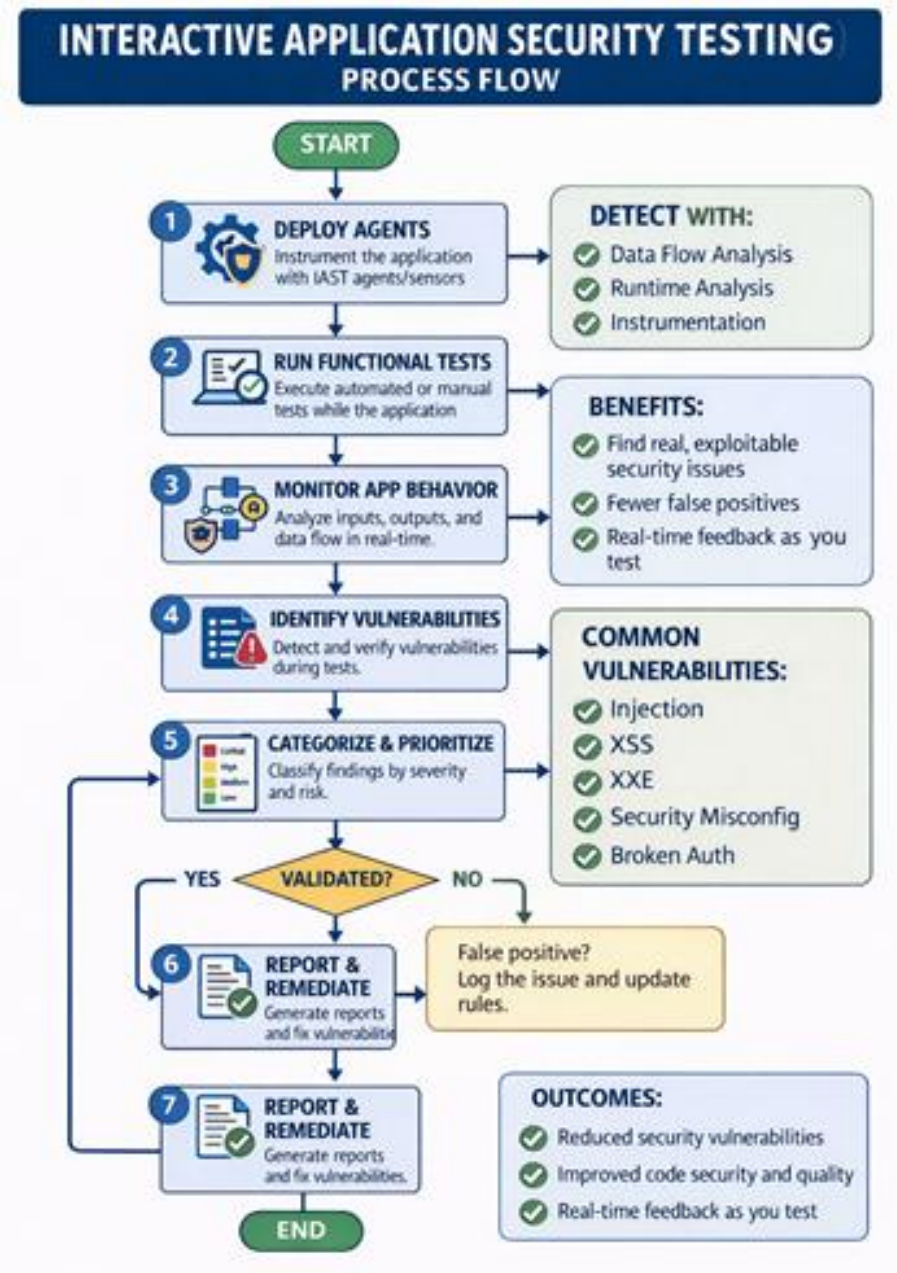
- VCG is an automated code security review tool for C++, C#, VB, PHP, Java and PL/SQL which is intended to drastically speed up the code review process by **identifying bad/insecure code**
- Steps for SCA:
  - **Install** and **run** the VCG tool
  - Select the **language** of your project
  - Browse the **project/file** of your interest and click **Scan**



Source: *https://sourceforge.net*

Interactive Application Security Testing (IAST)

Interactive Application Security Testing (IAST) is a security testing approach that analyzes an application from within while it is running, combining elements of both static and dynamic testing. By instrumenting the application or using an agent, IAST monitors code execution in real time during functional testing, identifying vulnerabilities such as injection flaws, insecure configurations, and data exposure issues with high accuracy. Because it has visibility into both the source code and runtime behavior, IAST can pinpoint the exact location of vulnerabilities and provide detailed context for remediation. This makes it highly effective for integration into modern DevOps and CI/CD pipelines, enabling continuous security testing without significantly slowing down development.

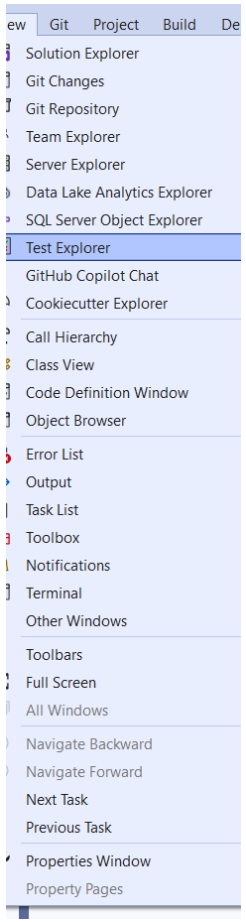


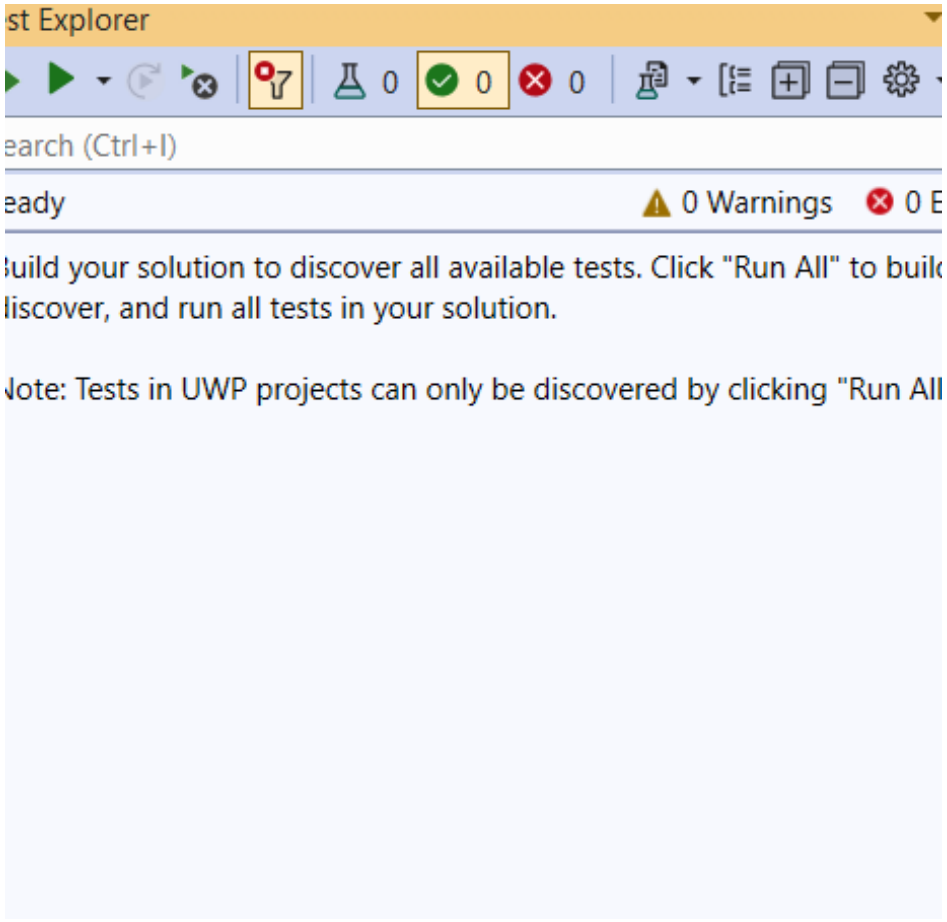
Interactive Application Security Testing (IAST

Active IAST vs. Passive IAST

	PASSIVE IAST	ACTIVE IAST
Speed of Assessing Vulnerability	Fast	Takes hours to days
Accuracy of Assessed Vulnerabilities	High	Low
Methodology of Vulnerability Assessment	Continuous	Based on scanning
DevSecOps Flexibility	High (applicable at any stage of software development lifecycle)	Low (applicable only at the end)
Authentication Flexibility	Yes (credentials are not required)	No (credentials are required)
Independent Deployment Model	Yes	No
Inside-out and Outside-in Visibility	Yes	No

Visual Studio Test Explorer





Software Technical Readiness Level (TRL)

TRL	Definition	Description	Supporting Information
1	Basic principles observed and reported.	Lowest level of software technology readiness. A new software domain is being investigated by the basic research community. This level extends to the development of basic use, basic properties of software architecture, mathematical formulations, and general algorithms.	Basic research activities, research articles, peer-reviewed white papers, point papers, early lab model of basic concept may be useful for substantiating the TRL.
2	Technology concept and/or application formulated.	Once basic principles are observed, practical applications can be invented. Applications are speculative, and there may be no proof or detailed analysis to support the assumptions. Examples are limited to analytic studies using synthetic data.	Applied research activities, analytic studies, small code units, and papers comparing competing technologies.
3	Analytical and experimental critical function and/or characteristic proof of concept.	Active R&D is initiated. The level at which scientific feasibility is demonstrated through analytical and laboratory studies. This level extends to the development of limited functionality environments to validate critical properties including cybersecurity and analytical predictions using non-integrated software components and partially representative data.	Algorithms run on a surrogate processor in a laboratory environment, instrumented components operating in a laboratory environment, laboratory results showing validation of critical properties.

DoD Software TRL Definitions, Descriptions, and Supporting Information

Software TRL

TRL	Definition	Description	Supporting Information
4	Module and/or subsystem validation in a laboratory environment (i.e., software prototype development environment).	Basic software components are integrated to establish that they will work together. They are relatively primitive with regard to efficiency and robustness compared with the eventual system. Architecture development initiated to include interoperability, reliability, maintainability, extensibility, scalability, and security issues. Emulation with current/legacy elements as appropriate. Prototypes developed to demonstrate different aspects of eventual system.	Advanced technology development, stand-alone prototype solving a synthetic full-scale problem, or stand-alone prototype processing fully representative data sets.
5	Module and/or subsystem validation in a relevant environment.	Level at which software technology is ready to start integration with existing systems. The prototype implementations conform to target environment/interfaces. Experiments with realistic problems. Simulated interfaces to existing systems. System software architecture established. Algorithms run on a processor(s) with characteristics expected in the operational environment.	System architecture diagram around technology element with critical performance requirements defined. Processor selection analysis, Simulation/Stimulation (Sim/Stim) Laboratory buildup plan. Software placed under configuration management. Commercial-of-the-shelf/ government-off-the-shelf (COTS/GOTS) components in the system software architecture are identified.
6	Module and/or subsystem validation in a relevant end-to-end environment.	Level at which the engineering feasibility of a software technology is demonstrated. This level extends to laboratory prototype implementations on full-scale realistic problems in which the software technology is partially integrated with existing hardware/software systems. Cybersecurity verification should be included in the testing.	Results from laboratory testing of a prototype package that is near the desired configuration in terms of performance, including physical, logical, data, and security interfaces. Comparisons between tested environment and operational environment analytically understood. Analysis and test measurements quantifying contribution to system-wide requirements such as throughput, scalability, and reliability. Analysis of human-computer (user environment) begun.

DoD Software TRL Definitions, Descriptions, and Supporting Information

Software TRL

TRL	Definition	Description	Supporting Information
7	System prototype demonstration in an operational, high-fidelity environment.	Level at which the program feasibility of a software technology is demonstrated. This level extends to operational environment prototype implementations, where critical technical risk functionality is available for demonstration and a test in which the software technology is well integrated with operational hardware/software systems.	Critical technological properties, including cybersecurity, are measured against requirements in an operational environment.
8	Actual system completed and mission qualified through test and demonstration in an operational environment.	Level at which a software technology is fully integrated with operational hardware and software systems. Software development documentation is complete. All functionality and cybersecurity measures tested in simulated and operational scenarios.	Published documentation and product technology refresh build schedule. Software resource reserve measured and tracked. All severity 1 and severity 2 defects are resolved/confirmed, and a reasonably low level of severity 3 defects remain open.
9	Actual system proven through successful mission-proven operational capabilities.	Level at which a software technology is readily repeatable and reusable. The software based on the technology is fully integrated with operational hardware/software systems. All software documentation verified. Successful operational experience. Sustaining software engineering support in place. Actual system.	Production configuration management reports. Defect resolution system and process is in place for deployed software to address defects discovered in production.

DoD Software TRL Definitions, Descriptions, and Supporting Information

Continuous Monitoring

Why Real-Time Monitoring Matters:
Proactive Threat Detection: Detects security breaches or vulnerabilities as they occur, allowing teams to respond promptly.
Minimized Impact: Reduces the potential damage by containing threats in their early stages.
Compliance Support: Helps maintain logs and records for regulatory and compliance requirements.
Enhanced Resilience: Strengthens the overall security posture by continuously assessing system health and detecting deviations.
-Learning, Maxwell. Advanced DevSecOps: A Guide to Advanced DevSecOps Practices .

OSSTMM

The Open-Source Security Testing Methodology Manual (OSSTMM) is a peer-reviewed framework for performing accurate and repeatable security testing. It's widely respected in both open-source and commercial environments and is published by the Institute for Security and Open Methodologies (ISECOM). The acronym RAV permeates OSSTMM

R – Real Security - The measurable level of control an organization has over its assets.
A – Actual Exposure - The extent to which a system or asset is exposed to unauthorized interactions.
V – Verified Trust - The level of trust that is based on verifiable controls rather than assumption or policy.

Version 3 https://www.isecom.org/OSSTMM.3.pdf

OSSTMM

These guidelines exist to assure the following:

1. The test was conducted thoroughly.

2. The test included all necessary channels.

3. The posture for the test complied with the law.

4. The results are measurable in a quantifiable way.

5. The results are consistent and repeatable.

6. The results contain only facts as derived from the tests themselves.

Version 3 https://www.isecom.org/OSSTMM.3.pdf

OSSTMM

Type		Description
1	Blind	The Analyst engages the target with no prior knowledge of its defenses, assets, or channels. The target is prepared for the audit, knowing in advance all the details of the audit. A blind audit primarily tests the skills of the Analyst. The breadth and depth of a blind audit can only be as vast as the Analyst's applicable knowledge and efficiency allows. In COMSEC and SPECSEC, this is often referred to as Ethical Hacking and in the PHYSSEC class, this is generally scripted as **War Gaming** or **Role Playing**.
2	Double Blind	The Analyst engages the target with no prior knowledge of its defenses, assets, or channels. The target is not notified in advance of the scope of the audit, the channels tested, or the test vectors. A double blind audit tests the skills of the Analyst and the preparedness of the target to unknown variables of agitation. The breadth and depth of any blind audit can only be as vast as the Analyst's applicable knowledge and efficiency allows. This is also known as a **Black Box test** or **Penetration test**.
3	Gray Box	The Analyst engages the target with limited knowledge of its defenses and assets and full knowledge of channels. The target is prepared for the audit, knowing in advance all the details of the audit. A gray box audit tests the skills of the Analyst. The nature of the test is efficiency. The breadth and depth depends upon the quality of the information provided to the Analyst before the test as well as the Analyst's applicable knowledge. This type of test is often referred to as a **Vulnerability Test** and is most often initiated by the target as a self-assessment.
4	Double Gray Box	The Analyst engages the target with limited knowledge of its defenses and assets and full knowledge of channels. The target is notified in advance of the scope and time frame of the audit but not the channels tested or the test vectors. A double gray box audit tests the skills of the Analyst and the target's preparedness to unknown variables of agitation. The breadth and depth depends upon the quality of the information provided to the Analyst and the target before the test as well as the Analyst's applicable knowledge. This is also known as a **White Box** test.
5	Tandem	The Analyst and the target are prepared for the audit, both knowing in advance all the details of the audit. A tandem audit tests the protection and controls of the target. However, it cannot test the preparedness of the target to unknown variables of agitation. The true nature of the test is thoroughness as the Analyst does have full view of all tests and their responses. The breadth and depth depends upon the quality of the information provided to the Analyst before the test (transparency) as well as the Analyst's applicable knowledge. This is often known as an In-House Audit or a **Crystal Box** test and the Analyst is often part of the security process.
6	Reversal	The Analyst engages the target with full knowledge of its processes and operational security, but the target knows nothing of what, how, or when the Analyst will be testing. The true nature of this test is to audit the preparedness of the target to unknown variables and vectors of agitation. The breadth and depth depends upon the quality of the information provided to the Analyst and the Analyst's applicable knowledge and creativity. This is also often called a **Red Team exercise**.

OSSTMM - Errors

Error Type		Description
1	False Positive	*Something determined as true is actually revealed false.* The target response indicates a particular state as true although in reality the state is not true. A false positive often occurs when the Analyst's expectations or assumptions of what indicates a particular state do not hold to real-world conditions which are rarely black and white.
2	False Negative	*Something determined as false is actually revealed as true.* The target response indicates a particular state as not true although in reality the state is true. A false negative often occurs when the Analyst's expectations or assumptions about the target do not hold to real-world conditions, the tools are not adequate for the test, the tools are misused, or the Analyst lacks experience. A false negative can be dangerous as it is a misdiagnosis of a secure state when it does not exist.
3	Gray Positive	*Something answers true to everything even if false.* The target response indicates a particular state as true, however the target is designed to respond to any cause with this state whether it is true or not. This type of security through obscurity may be dangerous, as the illusion cannot be guaranteed to work the same for all stimuli.
4	Gray Negative	*Something answers false to everything even if true.* The target response indicates a particular state as not true, however the target is designed to respond to any cause with this state whether it is true or not. This type of security through obscurity may be dangerous, as the illusion cannot be guaranteed to work the same for all stimuli.

OSSTMM - Errors

Error Type		Description
5	Specter	*Something answers either true or false but the real state is revealed as unknown.* The target response indicates a particular state as either true or false although in reality the state cannot be known. A specter often occurs when the Analyst receives a response from an external stimulus that is perceived to be from the target. A specter may be intentional, an anomaly from within the channel, or the result of carelessness or inexperience from the Analyst. One of the most common problems in the echo process is the assumption that the response is a result of the test. Cause and effect testing in the real world cannot achieve consistently reliable results since neither the cause nor the effect can be properly isolated.
6	Indiscretion	*Something answers either true or false depending when it's asked.* The target response indicates a particular state as either true or false but only during a particular time, which may or may not follow a pattern. If the response cannot be verified at a time when the state changes, it may prevent the Analyst from comprehending the other state. An Analyst may also determine that this is an anomaly or a problem with testing equipment, especially if the Analyst failed to calibrate the equipment prior to the test or perform appropriate logistics and controls. An indiscretion can be dangerous as it may lead to a false reporting of the state of security.
7	Entropy Error	*The answer is lost or confused in signal noise.* The target response cannot accurately indicate a particular state as either true or false due to a high noise to signal ratio. Akin to the idea of losing a flashlight beam in the sunlight, the Analyst cannot properly determine state until the noise is reduced. This type of environmentally caused error rarely exists in a lab, however it is a normal occurrence in an uncontrolled environment. Entropy can be dangerous, if its effects cannot be countered.
8	Falsification	*The answer changes depending on how and where the question is asked.* The target response indicates a particular state as either true or false although in reality the state is dependent upon largely unknown variables due to target bias. This type of security through obscurity may be dangerous, as the bias will shift when tests come from different vectors or employ different techniques. It is also likely that the target is not aware of the bias.

OSSTMM - Errors

9	Sampling Error	*The answer cannot represent the whole because the scope has been altered.* The target is a biased sample of a larger system or a larger number of possible states. This error normally occurs when an authority influences the operational state of the target for the duration of the test. This may be through specific time constraints on the test or a bias of testing only components designated as "important" within a system. This type of error will cause a misrepresentation of the overall operational security.
10	Constraint	*The answer changes depending on the limitations of the tools used.* The limitations of human senses or equipment capabilities indicate a particular state as either true or false although the actual state is unknown. This error is not caused by poor judgment or wrong equipment choices rather it is a failure to recognize imposed constraints or limitations.
11	Propagation	*The answer is presumed to be of one state or the other although no test was made.* The Analyst does not make a particular test or has a bias to ignore a particular result due to a presumed outcome. This is often a blinding from experience or a confirmation bias. The test may be repeated many times or the tools and equipment may be modified to have the desired outcome. As the name implies, a process that receives no feedback where the errors remain unknown or ignored will propagate further errors as the testing continues. Propagation errors may be dangerous because the errors propagated from early in testing may not be visible during an analysis of conclusions. Furthermore, a study of the entire test process is required to discover propagation errors.
12	Human Error	*The answer changes depending on the skill of the Analyst.* An error caused by lack of ability, experience, or comprehension is not one of bias and is always a factor that is present, regardless of methodology or technique. While an experienced Analyst may make propagation errors, one without experience is more likely to not recognize human error, something that experience teaches to recognize and compensate for. Statistically, there is an indirect relationship between experience and human error. The less experience an Analyst has, the greater the amount of human error an audit may contain.