FIT2107

Software Quality and Testing

# Quality Assurance

**Topics Covered**

Risk Software Quality

Software Quality Assurance Quality in Use

Product Quality Properties Verification and Validation

Software quality assurance is a risk management technique

## Risk

Risk factors are things that happen in a project that would be bad. In risk management, you figure out those factors and decide how likely they are and how bad the consequences are.

Based on this you can decide how to handle the risk, this could be from doing nothing to investing in preventing the risk and damage.

## Software Quality

Quality software is software that does what the customer needs. The fullness of what the customer actually needs and the requirements given is the quality.

So quality is about meeting stated and implied customer needs.

Requirements may change so while quality is about the behaviour on initial delivery it also encompasses how that software can change.

## Software Quality Assurance

Software quality assurance is a way of reducing some of the risks associated with software development.

* **Artifacts:** The identifiable outputs of a project. This could include
  + Source code
  + Documentation
  + Reports
* **Deliverables:** The artifacts delivered to a customer
  + Executable software
  + User documentation
  + Requirements documentation

At a high level, SQA is designed to mitigate risks that artifacts are not of sufficient quality.

Many consequences can occur as a result of deliverables being insufficient in quality, but many consequences can be treated back to undelivered artifacts as well.

SQA is how one measures that the project’s quality to ensure its up to snuff. It’s also about ensuring the software processes are appropriate to create our product.

QA involves checking if

* The product is of sufficient quality
* The process to produce the product is sufficient in quality
* The process of checking the first two is comprehensive and documented

## Limits of SQA

* Inability to prove correctness
* Inability to remove all risk
* Too much work to assure everything
* Everyone should be involved in SQA

## Quality in Use

Different users can have different views on the quality of the exact same product. Quality in use is the degree to which a particular user’s needs are met, whether it puts them in risk, and its emotional effect.

Therefore, we work with a range of stakeholders to develop quality goals.

## Product Quality Properties

* **Functional Stability:** How well a product meets needs
  + Functional completeness
  + Functional correctness
  + Functional appropriateness
* **Performance Efficiency:** Performance under conditions
  + Time behaviour
  + Resource utilisation
  + Capacity
* **Compatibility:** Performance within different hardware and software environments
  + Co-existence
  + Interoperability
* **Usability:** How effectively and efficiently a user can meet their needs
  + Appropriateness recognisability
  + Learnability
  + Operability
  + User Error Protection
  + User interface aesthetics
  + Accessibility
* **Reliability:** Performance under conditions
  + Maturity
  + Availability
  + Fault tolerance
  + Recoverability
* **Security:** How well information is protected
  + Confidentiality
  + Integrity
  + Non-repudiation
  + Accountability
  + Authenticity
* **Maintainability:** How easily and effectively it can be modified
  + Modularity
  + Reusability
  + Analysability
  + Modifiability
  + Testability
* **Portability:** How well it can be transferred between environments
  + Adaptability
  + Installability
  + Replaceability

Requirements documents rarely cover all these, as they’re not equally important in software projects. Finding relevant properties is important to achieving quality.

## Verification and Validation

* **Verification:** Are we building the product right
* **Validation:** Are we building the right product

## Artifact Verification

Any artifact can be verified, however, the methods can vary

* **Executable Artifacts:** Source code can be run and tested so their behaviour is observed.
* **Formal Artifacts:** Those which have rigorous an unambiguous definitions of their syntax and definition. Formal artifacts can be verified using mathematical techniques.
* **Informal Artifacts:** These include everything else. These can’t be tested or proved so verification is done manually.

# Quality Plans and Methods

**Topics Covered**

IEEE Standard for SQAP Sections of the IEEE SQAP

There must be a systematic approach to SQA activities, else some artifacts may be neglected. First, we identify the quality requirements, then we specify the activities, processes, and standards to employ, then justify their sufficiency.

## Software Quality Assurance Plan (SQAP)

This exists so everyone in a project known how SQA will occur. It’s documented such that

* It’s not subject to opinion
* A judgement can be made that it’s accurate
* We can assess whether SQA is done as planned

So the SQAP defines the process for ensuring the product being built meets defined quality requirements.

## IEEE Standard for SQAP

The IEEE defines a standard format for the SQAP. Projects can have “full” conformance or “tailored” conformance to their standard. It rarely needs to be fully conformed to but it provides a structure that is comprehensive and easy to follow.

The SQAP you create should answer

* Who is responsible for answering SQA activities?
* What SQA activities need to be performed on what artifacts, and what processes?
* Who needs to be involved in individual SQA activities?
* How extensive should SQA activities be?
* What and how will software quality be measured?
* How will the results of SQA be made available to relevant stakeholders?

## Sections of the IEEE SQAP

**Preliminaries**

1. **Purpose and Scope:** Short and sweet. Say what the SQAP is for and the project it applies to.
2. **Definitions and Acronyms:** Put the project-specific terms or abbreviations used in the document
3. **Reference Documents:** Specific and detailed references used in the document may include
   1. A project does (plans and standards)
   2. A company does (policies and standards)
   3. Industry standards and references

## SQA Plan Overview

1. **SQA Plan Overview**
   1. **Organisation and Independence:** Who’s responsible for SQA, and their relationship to the management of the project
   2. **Software Product Risk:** Describe product risks; the risks associated with the use of the software product itself
   3. **Tools:** List the tools that will be used for SQA
   4. **Standards, Practices, and Conventions:** Industry-specific standards, practices, and conventions, especially used in high-risk industries
   5. **Effort, Resources, and Schedule:** State the resources required for SQA, their specific jobs, estimate the effort in time, and provide a schedule of SQA tasks

## Activities, Outcomes and Tasks

1. **Activities, Outcomes and Tasks**
   1. **Product Assurance:** Explain how you’re going to ensure the product meets quality goals
      1. **Evaluate Plans for Conformance:** Describe how to ensure the product meets quality goals. Includes
         * Requirements Documents
         * High-Level Design Documents
         * Plans and Schedules
      2. **Evaluate Product for Conformance:** Describe how to verify the product is in accordance with requirements and design
      3. **Evaluate Products for Acceptability:** Describe how to determine if the product is acceptable for customers
      4. **Evaluate Life Cycle Support for Conformance:** Plan how to check whether post-release support is sufficient to any contracts or agreements with the customer
      5. **Measure Products:** List what measurements to use to assess the quality of the product, and how they’ll be collected and used.
   2. **Process Assurance:** Describe how to ensure the process by which you build the product conforms to plans
      1. **Evaluate Life Cycle Processes for Conformance:** Describe how to assess whether the life cycle process in the management plan is reasonable and actual processes match management plan
      2. **Evaluate Environments for Conformance:** Describe how to ensure programming and test environments match the requirements and management plans
      3. **Evaluate Subcontracting Processes for Conformance:** Parts of development may be subcontracted to third parties. It may be necessary to pass on process requirements, this section describes how that is achieved
      4. **Measure Processes:** Describe how to ensure process metrics are appropriate to the project and how to ensure measurement activities are done according to plans
      5. **Assess Staff Skill and Knowledge:** Describe how to assess whether staff assigned have the right mix of skill and knowledge to complete the project

## Additional Considerations

1. **Additional Considerations**
   1. **Contract Review:** If developed based on a contract, the contract needs to be quality assured

## Quality Measurements

* 1. **Waivers and Deviations:** Describe the process for modifying the SQA process
  2. **Task Repetition:** Describe conditions where previously completed SQA tasks need to be repeated
  3. **Risks to Performing SQA:** A risk analysis for the SQA activities
  4. **Communications Strategy:** Describe how the results of SQA will be reported to stakeholders

## Non-Conformance Processes

**SQA Records**

1. **SQA Records:** Describe how record keeping is performed, including

* What records are kept
* What they’ll contain
* How they’ll be stored
* Who’s responsible for maintaining them
* How they’re accessed and who has access
* How long they’re kept and who disposes of them

## b. Analyse, Identify, Collect, File, Maintain and Dispose

**c. Availability of Records**

# Reviews, Inspections, and Walkthroughs

**Topics Covered** Human Inspection Fagan Inspections Alternative Inspections

## Human Inspection

In practice, the most useful family of techniques for improving software quality is human inspection. Inspections tend to find different faults to execution based testing.

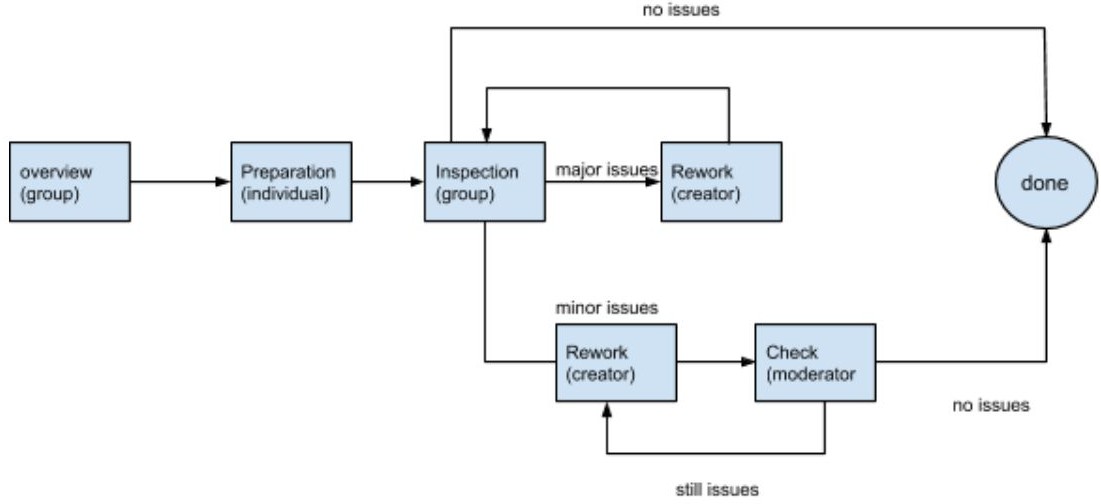
Maintainability and portability are two software quality properties that are best assessed with human review.

Generally, the later a mistake is found the more costly it will be to fix. The way to catch these mistakes is through human review.

## Fagan Inspections

How to conduct Fagan inspections

1. **Who:** Involves four participants
   1. **Moderator:** A developer
   2. **Designer:** The person who designed the feature
   3. **Coder:** The person who wrote the code
   4. **Tester:** The person writing test cases
2. **What:** The flowchart below describes Fagan inspection



* 1. **Overview:** The designer explains the design of the component under review to the participants and provides the artifact to the inspection team
  2. **Preparation:** Review participants study the artifact to understand and find problems. They’re also provided with
     + A list of common problems in recent inspections
     + A checklist of ways to find problems
  3. **Inspection:** Participants meet and explain the artifact in detail and answer questions to find problems
  4. **Rework:** The person responsible for the artifact fixes the error

## Follow-Up

* **Minor Problems:** The moderator can check issues themselves or reconvene the inspection panel
* **Major Problems:** The inspection panel must be reconvened

## What Should be on the Checklists

* 1. **Notation:** Does the artifact follow system rules?
  2. **Completeness:** Is there everything that needs to be there?
  3. **Testability:** Is there a way to test it?
  4. **Consistency:** Is there consistency between artifacts?
  5. **Error Handling:** Are problems taken care of?
  6. **Clearly Identified Purpose:** What is this for?
  7. **Writing Quality:** Are there any English errors?

## Overhead

Fagan inspection meetings are quite costly

* Requires all four members to be at the same place at the same time
* Most defects are found in preparation, not inspection
* Most faults are found by a single reviewer
* Teams that use tools for inspections without meeting can find as many faults as Fagan teams
* Physical inspections may result in fewer false positives

## Alternative Inspections

Fagan’s key features are

* Involving a team
* Conducted in-person
* Checklists provide things to do
* A formal process for approval

These features can be tailored or omitted for your needs

## Tool Based Support for Nominal Inspections

Software Gits can assist with reviews made in person

## Instantaneous Reviewing With Pair Programming

Pair programming is an agile technique which gives an instantaneous code review

# Testing Strategies

**Topics Covered** Objectives Test Levels

Functional Correctness Testing

## Software Testing

The dynamic verification that a program provides expected behaviours on a finite set of test cases, selected from a domain. In short, executing software to ensure it functions correctly.

## Objectives

* **Functional Correctness:** Ensuring the software produces the outputs consistent with the specifications, given the input.
* **User Acceptance:** Ensuring the system as a whole meets user’s high-level requirements.
* **Performance:** Ensuring the software meets resource utilisation requirements.
* **Security:** Another form of functional correctness, but aiming to ensure security risks are minimal. Fuzz and penetration testing are used.
* **Usability:** Ensuring the software enables people to achieve their goals.
* **Reliability:** Assessing how regularly the software fails to perform as specified
* **Robustness:** Looking into the consequence of breakage (as opposed to the reason)
* **Regression:** Ensuring a system hasn’t changed after modification

## Test Levels

* **Unit Testing:** Testing, in isolation, the smallest separately testable components of a software system, usually a class.
* **Integration Testing:** Testing interactions between components within a system, usually conducted after unit testing
* **System Testing:** Involves testing the entire system

## Functional Correctness Testing

* **Selecting Test Cases:** The aim of testing isn’t to find the most faults. It’s to find the most critical faults. The following techniques can achieve this
* **Exploratory Testing:** Described as simultaneous learning, test design, and test execution. While the software is being tested the tester learns new good tests to run
* **Systematic Testing:** Deciding, in advance, a strategy to select a set of test cases
  + **Black Box:** Constructing tests based on what the software is to do
  + **White Box:** Constructing test cases based on the structure of the code

# Black Box Testing

**Topics Covered** Random Testing Equivalence Partitioning Combinatorial Testing

Other Black Box Methods

## Random Testing

Using a random generator to select data for input data

## Equivalence Partitioning

All software testing that isn’t random is seen as subdomain testing. Subdomain testing is one that divides the input domain into a subdomain and selects cases from each subdomain.

Subdomain testing methods vary by

* How domain is divided into subdomains
* How test cases are allocated to each subdomain

A division of the domain into a number of disjointed subdomains, such that the domain is the union of subdomains, is a partition. The partition is an equivalence partition if all elements in a subdomain are equivalent in some way. The subdomains of equivalence partitions are equivalence classes.

## Partition Testing

Partitions testing is most effective when their partitions have homogeneous behaviour, that is, either every input works in a partition, or fails.

To conduct equivalence partition testing you must

* Identify the input domain of the program, including valid and invalid
* Based on experience, identify parts that the program will treat similarly and therefore will have homogeneous behaviour
* Write down those equivalence partitions
* From each partition pick a number of tests
* For each test input, know what the expected output is
* Test!

## Finding Equivalence Classes

* **Numerical Ranges:** Found on ranged inputs or list lengths
* **Categories:** Inputs that fall into defined categories, like odd or even
* **Invalid Inputs:** Inputs that don’t make sense for a certain goal, often positive or negative
* **Looking at Outputs:** Looking at the outputs and working to inputs

## Multiple Properties - The Category-Partition Method

Often there will be multiple properties on which you can define equivalence classes. This can lead to an explosion of equivalence classes. Rather than going straight to equivalence classes you apply a process to generate them

1. Identify what affects the behaviour of the part being tested into categories
2. Partition the category into choices - the equivalence partitions of each category
3. List all combinations of categories and their choices
4. Eliminate irrelevant or duplicate situations
5. Generate test inputs
6. Test!

Keep in mind test cases must

* Be unambiguous
* Be complete
* Have expected behaviour documented

## Combinatorial Testing

Combinatorial explosions are likely to occur, n categories with at least m choices have at least mn test frames, which could be a lot. Combinatorial testing helps with this.

What if we ignored the combinations and instead ensured choices for given categories was tested in at least one test case. Higher “iterations” can be tested too, so from the case of two interactions we consider every pair of categories so that each pair is tested in at least one case.

## Pairwise Coverage

A good number of interactions is two, where we consider every pair of categories. We ensure to have each combination of every choice in two categories.

## Boundary Value Analysis

In practice, errors tend to cluster around partition boundaries, though remember equivalence partition testing is based on the idea that tests within a partition are equal, so don’t rely just on boundary testing.

## Domain Testing

1. For inequalities, choose a point on the boundary and another close to the boundary
2. For equalities, choose a point on the boundary and two points on either side of the boundary

## Other Black Box Methods

* Decision Table Testing
* Model-Based Testing
* Ad-Hoc Testing

# Documenting Testing & Tracking Issues

**Topics Covered**

IEEE Software Testing Standard Test Documentation

Bug Reporting & Issue Tracking

You’re going to want to plan and document testing to be confident in your testing. If you’re paying a third party to perform tests, you’ll need to document the testing to ensure it’s adequate.

## IEEE Software Testing Standard

The IEEE 29119 Software Testing Standard can be broken down into five parts

1. Concepts & Definition
2. Test Processes
3. Test Documentation
4. Test Techniques
5. Keyword Domain Testing

## Test Documentation

There are many different places in which testing and planning is documented

## Organisational Test Policies

In large organisations, there may be a place to devise general policies that are applied access projects. This can save effort in preparing project-specific plans.

## Project Test Plans

This is where you set out the plan for testing a specific project. It should address the following points

* **Risks to be Addressed:** What bad things are we avoiding with this testing
* **Testing Strategy:** What strategies to use based on risks and resources
* **Testing Artifacts:** What will be produced by testing
* **Testing Environment:** What tests, software and hardware to use
* **Personnel:** Who’s responsible for tests
* **Scheduling:** When will testing be conducted
* **Results Reporting:** How will results be made available to the relevant people
* **Metrics:** What metrics will you collect

## Test Cases and Test Procedures

The three most important documents to specify low-level test processes are

* Test environment documents to describe the required software and hardware
* Test procedures to describe setup procedures and a set of test cases
* Test cases documents contain the input data and expected results

There are some key things the IEEE asks you to document in these test cases

* Unique IDs for tests
* Scope of tests
* Objectives of tests
* Tracing tests back to the relevant software requirement
* Who needs the results
* Description of test cases including
  + Preconditions
  + Inputs
  + A detailed description of expected results

## Traceability Matrix

This is a document used to record the relationship between test cases and software requirements. It is a table relating a requirement ID to a test ID.

## Test Case Results

You should record

* What tests were done
* When they were done
* Who did them
* Any deviations from procedure
* Results of testing
* Signoff by relevant people

## Bug Reporting & Issue Tracking

**Tracking Issues**

There must be a mechanism to track whether a fault is identified, diagnosed, or fixed. Some may also turn out to be false positives

* **Bug Reporting:** Any bug you report should contain all the information necessary to reproduce the fault. If you’re unable to reproduce it, you should still give as much information as possible. With enough reports from many people, a developer can piece together a problem.
* **Issue Tracking Tools:** Issues have a lot of info attached to them, with many issues per project. To manage this information tracking tools have been developed with features including
  + An interface for reporting and tracking issues
  + Assign people responsibility
  + Classify issues
  + Workflow management
  + A query tool to search and report specific issues
  + Reporting tools to examine the whole project
* **Issues Workflows:** For a given project, the team will define an issue lifecycle, where issues have a status assigned to them
* **New:** When an issue is first reported. Typically assigned to someone and given a priority
* **In Progress:** Someone is working on it
* **Testing:** A decision is made to fix the bug
* **Regression Testing:** When a bug is found a test to expose the bug is created and added to the pool of tests
  + These tests are run when a change is made to ensure any past bugs don’t return

# White Box Testing

**Topics Covered** Statement Coverage Branch Coverage Control Flow Diagram

Condition Based Coverage Criteria Path Coverage

Data Flow Coverage Dealing With Exceptions Uses of Coverage Criteria

In structural testing or white box testing, you select your test cases to ensure source code is systematically covered by test cases. We cover the behaviour of the system as it exists and not solely on the specification.

## Statement Coverage

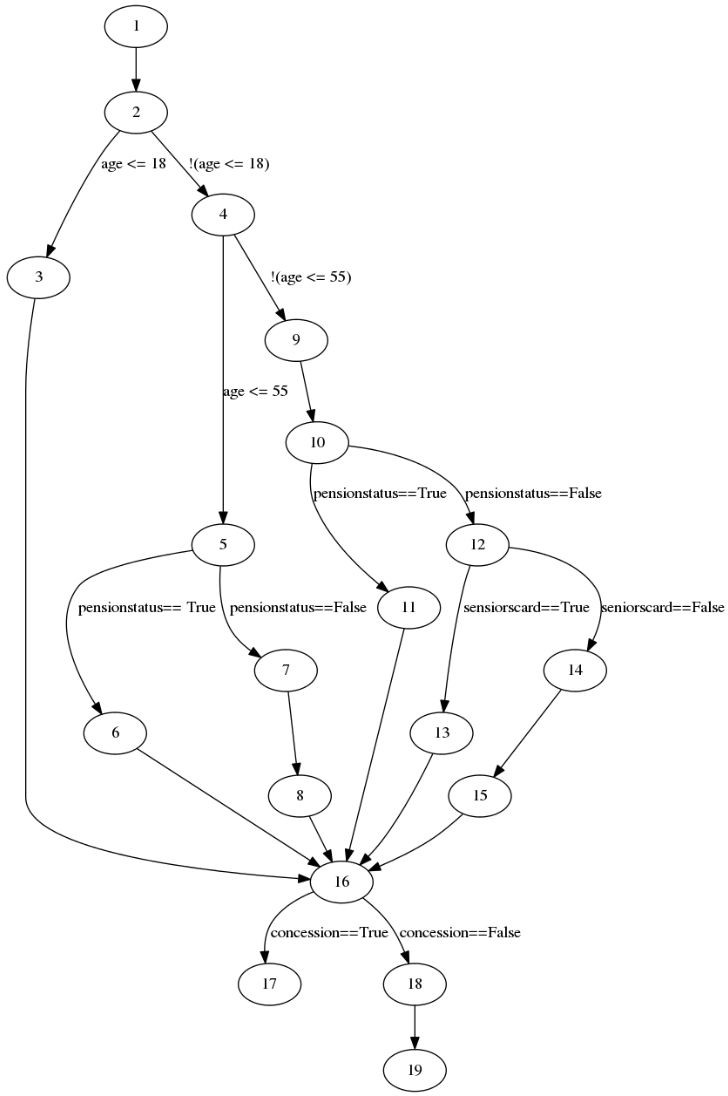
To perform coverage testing, you must select cases that meet the following criterion: “All statements in a program should be executed at least once”, though this isn’t always possible, larger projects aim for a high percentage of coverage.

This method generally isn’t sufficient but it prevents mutants and allows for a more effective slaying of mutants.

## Branch Coverage

In branch coverage you must write enough test cases that each decision has a true and false outcome at least once, in other words, each branch direction must be traversed at least once. This includes ifs, whiles, and non-trivial fors.

## Control Flow Graphs

Statements can be viewed as nodes in a graph and the program can jump between them as edges.

The node numbers represent line numbers

## Branch Coverage and CFGs

Branch coverage is equivalent to having each edge of a CFG traversed at least once.

## Finding Tests for Branch Coverage

1. Identify decision nodes
2. For each decision node, identify the edges coming out and the paths that get you there
3. For the paths to the node, write down the path constraints that the variables have to take to get you there
4. For each edge coming out write the constraints for that edge
5. Find an input that leads to the edge, this is the test case
   * If you find an input record it as a test case
   * If you can’t find an input it is infeasible
6. Repeat until all test cases are found

## The Cheat’s Way

In practice you don’t need to draw a CFG, all you need to do is identify where branches and as what input you need to reach the branch and get a condition

## How Many Test Cases?

The upper bound of required test cases, M, is given by

M = (number of branch points) + 1

## Condition Based Coverage Criteria

**Condition Coverage**

The naive way to tackle compound predicates is condition coverage, which says every simple boolean expression should be covered in a case that it’s true and one that it’s false.

Unfortunately, it’s not useful on its own, we have the notion of subsumptions. For example

x and y

Has condition coverage by x = true, y = false x = false, y = true

But these both have the same result, it has no decision coverage

## Condition/Decision Coverage

One solution is to combine condition and decision coverage. You need to select cases that achieve both coverages, for example

true and true false and false

## Modified Condition/Decision Coverage

MD/DC imposes two additional requirements

1. Every point of entry and exit in the program occurs at least once
2. Every condition in a decision has been shown to inadequately affect the decision’s outcome

## Multiple Condition Coverage

This is a condition coverage which requires every possible combination of true and false values for each condition within a decision.

## Path Coverage

There are some faults which branch coverage can miss, where a certain decision causes a bug in later decisions. In path coverage, you must cover all execution paths through a system under test.

This can be difficult, however, as

* Some paths are infeasible
* The number of tests required can increase quickly
* Loops may cause an infinite number of required paths

## Data Flow Coverage

This involves looking at data definitions and uses in the software rather than control flow. You look at places in your software where variable values can be defined and used and select cases covering all possible cases.

## Dealing With Exceptions

Exceptions can vary execution paths but don’t fit in well with coverage criteria. Any method or operator that throws exceptions can create a branch point, leading to too many test branches.

## Uses of Coverage Criteria

There are two ways of using coverage criteria for testing

1. Systematically write a set of test cases for high amounts of the coverage criterion of your choice
2. Assess the quality of a test set by seeing how high the test coverage is

# Unit and Integration Testing

**Topics Covered** Automated Unit Testing Expected Exceptions Side Effects

Running the Tests Evaluating the Suite Integration Testing Continuous Integration

A well-designed software system will be composed of units, self-contained chunks of code available through an interface. Units are, essentially, classes.

Unit tests exercise the functionality of a unit, they’re typically written by the coder to automate testing. Integration testing exercises the integration of the unit within a system, so you test the bits of the system you integrate with.

## Automated Unit Testing

Unit testing frameworks exist to create unit tests automatically. Test cases are created by adding methods to a test class with the basic pattern

* Test something
* Assert it

## Unit Tests in Python

import unittest import Module

class ModuleTest(unittest.TestCase): “””

Tests for the Module class “””

def setUpClass(self):

# Only runs once for a test class shared\_variable = 0

def setUp(self):

# Called before each “test\_” method in the class is run self.module = Module()

def test\_module\_unit(self):

# methods with test\_ prefix are run for unittest.main() return\_value = self.module.module\_unit() assertCondition(return\_value)

if name == ‘ main ’: unittest.main()

**Assertions in Python**

...

class ModuleTest(unittest.TestCase): def setUp(self):

...

def test\_assertions(self):

# raises assertionError if the value given isn’t True self.assertTrue(boolean)

# raises assertionError if the value given isn’t False self.assertFalse(boolean)

# raises assertionError if the exception given isn’t raised with self.assertRaises(Exception):

self.module.module\_unit()

# raises assertionError if the two values aren’t equal self.assertEqual(self.module.value1, self.module.value2)

...

## Mocking in Python

import unittest

from mock import MagicMock, patch import module\_file

from module\_file import Module

# Create a MagicMock object to replace the original module mock\_module\_import = MagicMock()

# Any accessor to the mock creates another MagicMock and sets it to that accessor

# mock\_module\_import.method becomes a MagicMock

# mock\_module\_import.method.object becomes a MagicMock mock\_module\_import.method().object

# mock\_module\_import.some\_value is given the value 3.14 mock\_module\_import.some\_value = 3.14

# mock\_module\_import.other\_method() returns “hello world” mock\_module\_import.other\_method.return\_value = “hello world”

# Mocks can be given side effects such as functions or exceptions def side\_effect\_func(arg):

return arg mock\_module\_import\_sf = MagicMock()

mock\_module\_import\_sf.side\_effect = side\_effect\_func

# @patch replaces the modules in your imported file with the mocks @patch(‘module\_file.some\_module’, mock\_module\_import)

class MockingTest(unittest.TestCase): def setUp(self):

self.module = module\_file

def test\_mock\_module\_import(self):

# The module should do what you told it to do in the mock self.assertEqual(self.module.some\_module(), 3.14)

# Calls to undefined parts of the mock should create more mocks assertTrue(isinstance(self.module.undefined, MagicMock))

@patch(‘module\_file.some\_module’, mock\_module\_import\_sf) def test\_mock\_side\_effects(self):

# The side effect will make the mock behave as you tell it assertEqual(self.module(“arg”), “arg”)

**Expected Exceptions**

Some units need to be checked that they throw exceptions under certain conditions. In Python use with self.assertException(Exception): to check that they’re thrown as expected.

## Side Effects

Usually, we’d like to test units in isolation, but they may interact with the system or I/O. To test this without concern for the unpredictability of the outside world we replace things outside the unit with dummy code.

In Python use mock to create mocks directly or use patching to replace classes or methods with mocks.

## Mocking Effort vs Reward

Two extreme viewpoints exist; mock everything or mock nothing at all.

When you mock, you’re purposely not testing if the mock is upholding its end of the bargain for simplicity.

Mocking is best used when real use is difficult, unpredictable, or impossible.

## Running the Tests

The point of having tests is to run them to expose faults. Tests should report a failure and continue running.

## Evaluating the Suite

You can use the Python coverage model to collect statement and branch coverage from your suite.

## Integration Testing

Integration and integration testing are peas in a pod. Without one, the other is useless.

## Big Bang Integration

Write each module separately and unit test them. Once done, stick them all together and test the system

* An unprecedented amount of problems interacting with each other over a huge system

## Top-Down Testing

In a class diagram context, start with the top module (those with no in-arrows). Use stubs for those which the module connects to. Once tested, replace the stubs with their modules and top-down test each of those.

## Bottom-Up Testing

In a class diagram context, start with the bottom module (those with no out-arrows). Integrate them with their parents, and once tested, do the bottom up method on the parents.

## Sandwich Testing

Do bottom-up and top-down testing until you reach the middle

## Problems

These methods prevent many problems, including

* Not having a working system until completion
* Masses of repetitive testing and glue code
* Increasingly complex in complex systems

## Continuous Integration

* Keep your code in a repository
* Everyone frequently commits to the master
* The master is always runnable
* Every commit triggers a build
* All builds trigger tests
* Fix anything that breaks the build
* Test on something that matches the production environment
* Make the resulting executable easy to install

## Continuous Integration Servers With Gitlab

Gitlab is a software development platform with version control, issue tracking, code review, CI/CD, and more. You can self-host Gitlab on your own server or a cloud provider, or it’s own Continuous Integration server.

Once set up, you can use any git client with Gitlab.

CI in Gitlab is controlled by the .gitlab-ci.yml file in your repository. The results are visible through the main Gitlab web interface.

# Security QA

**Topics Covered** Motivation to Hacking Security Quality Assurance Security Testing

Hacking Principles & Resources Tools and Techniques

## Motivation to Hacking

**For-Profit**

* **Click Fraud:** Fake valuable clicks such as ads or music for money
* **Ransomware:** Encrypt and hold valuable data hostage for a ransom
* **Identity Fraud:** Use information to impersonate someone
* **Stolen Accounts:** Accounts that seem human are valuable to those who use bots for fraud

## Corporate Espionage

* Information can affect stocks
* Ideologically opposed

## Methods of Hacking

* **Unsecured Systems:** Many systems are left insecure and taken advantage of
* **Social Engineering:** Using non-technical methods to gather sensitive information
  + **Phishing:** Contacting users impersonating a body to gather information
  + **Whaling:** Phishing with knowledge of your personal details to appear more trustworthy
* **Physical Infrastructure:** Gain physical access to a device
* **Weak Passwords:** having an easily guessable password makes it easy to enter a system
* **Buffer Overflows:** Intentionally breaking buffers to get data otherwise hidden
* **Command Injection:** Giving input data that is processed invalidly, allowing code to be introduced
* **SQL Injection:** SQL code is given inputs which when unchecked can bypass security or change a site’s behaviour maliciously
* **Cross-Site Scripting (XSS):** JavaScript code is given in inputs which when unchecked can bypass security or change a site’s behaviour maliciously
* **Man-in-the-Middle:** Monitor unencrypted traffic and alter routing tables
* **Crypto Analysis:** Comparing known hashes or encryption values to reverse engineer data

## Security Quality Assurance

Security bugs are difficult to QA against, any bug can be exploited to the maximum extent.

## Software Development Lifecycles and Security

Some organisations have developed software engineering models with security in mind

## The Open Web Application Security Project

* Developer Guide
* Application Security Verification Standard (ASVS)
* Microsoft Security Development Lifecycle
* SDL for Agile

## Training

A fundamental aspect of developing secure software is ensuring everyone has appropriate training in software security

## Security in Requirements Gathering

* **Security Risk Analysis**
* To determine the extent of the SQA techniques the project needs, you need to determine what the security requirements are. You should consider
  + What resources the system has access to
  + What are the consequences of resources misuse
  + Who are authorised to access the resources of the system
  + Among these people, who is authorised to access and modify what
  + What other systems does your system share data and resources with
  + Whether the proposed high-level architecture and tasks are known to be risky

## Documenting Security Requirements

* Security is a type of non-functional requirement, so it’s not well captured in Agile.
  + Treat security as constraints to stories, so they need to be completed for a story to be done
  + SDL recommends gates to prevent releases with serious issues

## Design

There are some techniques that are known to be successful for reviewing the design for security

## Threat Mocking

* **Threat Modelling Diagrams**
* An annotated version of a data flow diagram
* A boundary represents boundaries between parts of the system
* Start thinking about threats that exist at each boundary using STRIDE
  + **Spoofing:** Authentication
  + **Tampering:** Integrity
  + **Repudiation:** Non-repudiation
  + **Information Disclosure:** Confidentiality
  + **Denial of Service:** Availability
  + **Elevation Privilege:** Authorisation

## Security Testing

**Unit Level Security Testing**

Developers should tests security requirements with unit testing. The OSWAP testing guide identifies security properties that should be tested

* Does invalid data get rejected by input validation
* Does output only produce valid data
* Are resources no longer used freed
* Is data that’s no longer used freed
* Is data that’s no longer required freed
* Can method calls be used to access data other than intended

## System Level Security Testing

Rather than checking input validation in an individual method

## Penetration Testing

An authorised attack on a computer system to evaluate the security of the system. OSWAP suggests a two-phased methodology

1. **Passive Phase:** Gather information
2. **Active Phase:** Attempt to identify and exploit security problems

Penetration testing has several stages

* **Pre-Engagement Interactions:** Agree on the scope of the pen test with the client.
* **Intelligence Gathering:** Information about the organisation and system being tested is gathered
* **Threat Modelling:** Find plausible threats to the organisation and system within the testing scope
* **Vulnerability Analysis:** Based on the knowledge gained in previous phases, potential vulnerabilities are identified
* **Exploitation:** The system is attacked within the agreed scope. Shouldn’t cause damage to the client
* **Post Exploitation:** The tester collects evidence of the exploitations they achieved to demonstrate vulnerabilities
* **Reporting:** The pen tester will report their findings to the client

## Organisation Penetration Testing

Attack the infrastructure of an entire organisation

## Hacking Principles and Resources

There are seven phases of a penetration test

* **Pre-Engagement Interactions:** Sit with the client and agree on the scope of penetration testing
* **Intelligence Gathering:** Gather information about the organisation and system you’re trying to penetrate. This information will provide tools about how to attack the system
* **Threat Modelling:** Identify possible threats to the organisation and systems within the scope of the test. In cooperation of the organisation analyse their business, information, and infrastructure.
* **Vulnerability Analysis:** Use the information to identify potential vulnerabilities. Vulnerability scanners can be used.
* **Exploitation:** Use the information acquired to bypass security.
* **Post Exploitation:** Collect evidence of the exploitation and what could be achieved
* **Reporting:** Testers report their findings to the client and staff of the system

## Tools and Techniques Port Scanners

A tool that systematically connects to a list of ports to indicate what services they run.

## Vulnerability Identification Scanners

A program designed to assess systems for known weaknesses and report them to the user.

## Exploits

Methods of taking advantage of vulnerabilities to bypass security.

## Payload

The thing you do with the exploit.

# Quality Metrics

**Topics Covered**

Design and Code Metrics Object-Oriented Design Metrics Fault Prediction

Maintainability

Rules of Thumb and BYO Models Taylorism

We would like to

* Predict aspects of the product we deliver before completion
* Assess the product or process to make decisions about what to do next

## Coverage

Coverage metrics tells us how much of a program has been tested by the test suite

## Mutation Analysis

Repeatedly seeding changes into the source code, running the modified code, and seeing if the test suite will report a fault

* A mutation score will be generated to indicate how good the test suite is
* Can be costly to run as the test suite may have to be run on thousands of mutants

## Design and Code Metrics

A set of proxy metrics that can come from analysing code or design artifacts. These can be used to indicate many things

## Code Size

This can measure the size of a codebase. You can do this by counting the lines of code (LOC), though this approach is limited. You’ll often need to consider comments, whitespace, and how dense or sparse the code is.

Generally, larger methods are more error-prone.

## Halstead’s Software Science Metrics

Halstead tried to abstract the details of programming languages basing metrics on the following properties

* n1 = The number of distinct operators
* n2 = The number of distinct operands
* N1 = The total number of operators
* N2 = The total number of operands Using the following definitions
* **Operators:** Arithmetic, equality, assignment, logic, control structures, function calls, array references, etc.
* **Operands:** Identifiers, literals, labels, function names, etc.

From this you can calculate the following metrics

Program Vocabulary:

Program Length:

n = n1 + n2

N = N1 + N2

Calculated Program Length: Ň = n1log2(n1) + n2log2(n2)

Volume: V = Nlog2(n)

Difficulty: D = (n1/2) \* (N2/n2)

Effort: E = D \* V

Implementation Time: T = E/18 *secs*

Bug Count: B = (E^(2/3))/3000

## McCabe’s Complexity Matrix

We compute the approximation of the cyclomatic complexity using the formule

M = τ + 1

Where τ is the number of decisions. M relates to the testability of code, with over

10-15 being dicey and over 30 being untestable and unmaintainable.

## Microsoft’s Maintainability Index

An index value from 0 to 100 that represents the ease of maintaining code devised from Halstead and McCabe ratings.

* 20-100 are good ratings
* 10-19 are moderate
* 0-9 are poor

For a project the formula is given as

MI = 171 - 3.42ln(avgE) - 0.23avgV(G) - 16.2ln(avgLOC) + 0.99(avgCL)

Where

* **avgE:** Average Halstead effort for each module
* **avgV(G):** Average McCabe complexity for each module
* **avgLOC:** Average lines of code for each module
* **avgCL:** Average number of comments for each module For an individual class the formula is given as

MI = max(0,(100 \* (171 - 3.2ln(V) - 0.23(V(G)) - 16.2ln(LOC)))/171)

Where

* **V:** Halstead Volume
* **V(G):** McCabe’s Complexity
* **LOC:** Lines of Code

## Object-Oriented Design Metrics

The proxy metrics so far have been low level, they don’t look at interaction within the system. The more interactions and the more complex those interactions the more testing required.

There are some metrics that look at the bigger picture to assess OO systems.

## Weighted Methods Per Class (WMPC)

For a class the weighted methods per class is given by

*n*

WMC = ∑ ci

*i*+1

For a class with methods M1 to Mn with each with a complexity of c1 to cn

This can predict the time and effort to develop and maintain the class

## Depth of Inheritance Tree (DIT)

Assuming you have a class diagram the DIT for a class is the number of transitions from a root node

## Number of Children (NOC)

The number of children which immediately inherit from a class in the context of a class diagram.

## Coupling Between Objects (CBO)

This is defined for an individual class as the number of classes it interacts with directly. This includes method invocations and access to public data.

High coupling makes it harder to design, build, test, debug, extend, and reuse classes

## Response for a Class (RCF)

Defined for a given class C as the size of the set

All the methods in C and all the methods invoked directly by methods in C A larger RCF the more complex the class and therefore, more complex to test and debug

## Lack of Cohesion in Methods (LCOM)

Cohesion is the closeness of the relationship between elements of a module.

P += 1

Q += 1

if two methods don’t share a common instance variable

if two methods share a common instance variable

Once all methods are compared to every other method

LCOM = P - Q if P > Q

= 0 if Q <= P

## Fault Prediction

Use a bunch of metrics as ‘Big Data’ to create models of faulty software so that it can predict the number of faults.

People have done this with methods including

* Static code measures
* Software process metrics
* Personnel based information

## Maintainability

The maintainability index was designed to predict maintainability. There are obvious issues, however

* Do developers perception of maintainability relate to actual maintenance properties
* Does a metric originally defined in C relate to newer languages

Studies show, however, the predictions for maintainability were not enough for broad applicability

## Rules of Thumb and BYO Mocks

Most software engineers can study artifacts and give assessments of quality, but we’re a long way from automating it.

Anomalous metrics are usually good indicators, though. So while it’s difficult to make general recommendations from metrics, they’re still worth recording. If you do that for many large projects you’ll find your own patterns connecting proxies with the things you need. This can be done by keeping metrics as the product is in development, and seeing how this relates to what you can about,

eg

* The number and severity of failures reported after release
* Time required to fix bugs
* Time to add new features

## Taylorism

The idea of managing the behaviour of workers producing something by systematic measurement. Frederick Winslow Taylor espoused the following

* Based on measurement, develop a “science” for each worker’s job, ie, a manual describing how they should be doing their work as efficiently as possible.
* “Scientifically” choose the best people for the job and train them in the methodology
* Work with employees to ensure they follow the process

## Misusing Metrics

Inappropriate metrics and making inappropriate decisions based on them can have unfortunate consequences