Testing and Test Case Design

Quality Assurance (QA)

- the process of ensuring that the software being built has the required levels of quality.
- = Validation + Verification

Validation: are you building the right system [requirements correct]

Verification: are you building the system right [implemented correctly]

E.g., ~~It is very important to clearly distinguish between validation and verification.~~ [Whether something belongs under validation or verification is not that important. **What is more important is that we do both.**]

Code review: Systematic examination of code with the intention of finding where the code can be improved.

- **Pull Request reviews** [GitHub | BitBucket]
- **In pair programming** implicit review of the code by the other member when working on the same code.
- **Formal inspections:** Members of the inspection team play various roles
  - **author** who create the artifact
  - **moderator**: planner and executor of meeting
  - **secretary**: recorder of the findings of the inspection
  - **inspector/reviewer** inspects/reviews the artifact

Static analysis: analysis of code without actually executing the code.

- can find useful information
  [unused variables
  unhandled exceptions
  style errors
  statistics]
- Most modern IDEs has inbuilt static analysis capabilities
- Higher-end static analyzers can perform more complex analysis such as <u>locating potential bugs, memory leaks, inefficient code structures</u>, etc.
  [CheckStyle, PMD, FindBugs]

Linters: a subset of static analyzers aiming to **locate areas** where the code can be made **'cleaner'**.

Dynamic Analysis: requires the code to be executed to gather additional information about the code

- Can used for measure **test converge.**

Formal verification

- uses **mathematical techniques to prove** the **correctness** of a program.
- can be used to **prove the absence of errors**
- more commonly used in **safety-critical software** such as flight control systems.

<u>**Disadvantages**</u>

- Only proves the compliance with the specification, **but not the actual utility** of the software.
- It requires highly specialized notations and knowledge [expensive technique to administer]

Testing can only prove the **presence** of errors

SUT: software under test

Testability: How easy it is to test an SUT. The higher the testability, the easier it is to achieve better quality software.

Regression: When modify a system, may result in some unintended and undesirable effects on the system.

Regression testing: Re-testing of the software to detect regressions. Can not be automated but automation is <u>highly recommended.</u>

Developer testing: Testing done by developers themselves

<u>Pros:</u>

- Locating the cause of a test case failure is difficult due to the larger search space
- Fixing a bug found during such testing could result in major rework
- One bug might 'hide' other bugs
- The delivery may have to be delayed

<u>Cons:</u>

- **Only test situations that he knows** to work (i.e. test it too 'gently').
- May be blind to his own mistakes (if he did not consider a certain combination of input while writing the code, it is possible for him to **miss it again during testing**).
- **Misunderstood** what the SUT is supposed to do in the first place.
- A developer may **lack the testing expertise**.

Unit testing: Testing individual units (methods, classes, subsystems, ...) to ensure each piece works correctly.

- Requires the unit to be tested in isolation → bugs in the dependencies cannot influence the test

Stub:

- has the same interface as the component it replaces
- implementation is so simple [unlikely to have bugs]
- mimics the responses of the component, but only for a limited set of predetermined inputs.
- does not know how to respond to any other inputs
- mimicked responses are hard-coded rather than computed or retrieved from elsewhere [database]

Why

- Stubs can isolate the SUT from its dependencies.
- Use a hybrid of unit+integration tests to **minimize** the need for stubs.

Integration testing: testing whether different parts of the software **work together** (i.e. integrates) as expected.

- **Not simply a case of repeating** the unit test cases
- using the actual dependencies (instead of stubs)
- Additional test cases focus on the interactions between the parts.

System testing: take the **whole system** and test it against the system specification.

- done by a testing team (also called a QA team).
- based on the specified **external behavior** of the system
- Sometimes, system tests **go beyond the bounds** defined in the specification. This is **useful** when testing that the system fails 'gracefully' when pushed beyond its limits.
- includes testing against **NFRs** too.

Alpha testing is performed by the **users**, under controlled conditions set by the software development team.

Beta testing is performed by a **selected subset of target users** of the system in their natural work setting.

Open beta release: Release of not-yet-production-quality-but-almost-there software to the general population.

Dogfooding: When creators use their own product

Scripted testing: First write a set of test cases based on the expected behavior of the SUT, and then perform testing based on that set of test cases.

- More systematic, and hence, likely to discover more bugs given sufficient time

Exploratory testing: Devise test cases on-the-fly, creating new test cases based on the results of the past test cases.

- Simultaneous learning, test design, and test execution
- Also known as reactive testing, error guessing technique, attack-based testing, and bug hunting.
- Success depends on tester's prior experience and intuition.
- may allow us to detect some problems in a **short time**
- it is **not prudent to use exploratory testing as the sole means of testing a critical system.** <u>[Use a mixture of Scripted Testing and Exploratory Testing]</u>

Acceptance testing (aka User Acceptance Testing (**UAT**): test the system to **ensure** it **meets the user requirements**.)

- Involves testing the whole system
- **comes after system testing**

System Testing	Acceptance Testing
Done against the system specification	Done against the requirements specification
Done by testers of the project team	Done by a team that represents the customer
Done on the development environment or a test bed	Done on the deployment site or on a close simulation of the deployment site
Both negative and positive test cases	More focus on positive test cases

- In **smaller** projects, the developers may do system testing as well, in addition to developer testing.
- System testing is **more extensive** than accept. Testing

Test Driver: code that 'drives' the SUT for the purpose of testing i.e. invoking the SUT with test inputs and verifying if the behavior is as expected.

- JUnit is a tool for automated testing of Java programs.

<u>GUI Testing</u> [TestFX | Visual Studio |Selenium (For Web)]

- Testing the GUI is **much harder** than testing the CLI (Command Line Interface) or API
- **Moving as much logic as possible** out of the GUI can make GUI testing easier.

Test coverage: Metric used to measure the extent to which testing exercises the code.

- **Function/method coverage:** base on function executed
- **Statement coverage:** based on the # of lines of code
- **Decision/branch coverage :** based on the decision points exercised
- **Condition coverage:** based on the **boolean sub-expressions**, each evaluated to **<u>both true and false (Need to cover 2 conditions)</u>** with different test cases. Condition coverage is not the same as the decision coverage.

- **Path coverage** measures coverage in terms of possible paths through a given part of the code executed. 100% means all possible paths have been executed. A commonly used notation for path analysis is called the **Control Flow Graph (CFG)**.
  Eg. enter -> 2 -> 3 -> 2 -> 3 -> exit
- **Entry/exit coverage:** measures coverage in terms of possible calls to and exits from the operations in the SUT.

Dependency injection: Process of 'injecting' objects to **replace current dependencies** with a different object. This is often used to **inject stubs to isolate the SUT** from its dependencies so that it can be tested in isolation.
Test-Driven Development(TDD) advocates writing the tests before writing the SUT, while evolving functionality and tests in small increments.

Test Cases Design

- Except for trivial SUTs, exhaustive testing is not practical
- Every test case adds to the cost of testing.
- Testing should be **effective** i.e., it finds a high percentage of existing bugs
  [Determine by absolute # of bugs detected]
- Testing should be **efficient** i.e., it has a high rate of success (bugs found/test cases)
  [Determine by **#bugs / #test cases**]
- For testing to be E&E, each new test you add should be targeting a potential fault that is not already targeted by existing test cases.

Positive test case: designed to produce an expected/valid behavior. E.g., Integer i == new Integer(50)

Negative test case: designed to produce a behavior that indicates an invalid/unexpected situation, such as an error message. E.g., Integer i == null

T**hree types**: based on how much of the **SUT's internal details** are considered when designing test cases:

- **Black-box (specification-based or responsibility-based) approach:** test cases are designed **exclusively** based on the SUT's specified external behavior.
- **White-box (glass-box or structured or implementation-based) approach:** test cases are designed **based on** what is known about the SUT's implementation.
- **Gray-box approach:** test case design uses some **important information** about the implementation.

Equivalence partition: A group of test inputs that are likely to be processed by the SUT in the same way.

- An EP may not have adjacent values.

By dividing possible inputs into **EPs** you can,

- avoid testing too many inputs from one partition
  increases the efficiency by reducing redundant cases.
- ensure all partitions are tested.
  increases the effectiveness by ↑ chance of finding bugs.

<u>**Example**</u>

Consider a Java method `isPrime(int i)` that returns true if i is a prime number.

'All non-int values' is a possible EP for testing this method.

False. As Java is strongly-typed, it is not even possible to use non-int values to test the method.

Boundary Value Analysis (BVA): test case design heuristic that is based on the observation that bugs often result from incorrect handling of boundaries of equivalence partitions.

- values near boundaries (i.e. boundary values) are **more likely** to find bugs, but not just test boundary values!
- Boundary values are sometimes called corner cases.
- Choose 3 values around the boundary to test: **boundary value, just below and just above** the boundary.

<u>**Combining test inputs**</u>**:** Testing all possible combinations is **effective** but **not efficient**. Here lists 4 types:

- **All combinations strategy** generates test cases for each unique combination of test inputs.
- **At least once strategy** includes each test input at least once.
- **All pairs strategy** creates test cases so that for any given pair of inputs, all combinations between them are tested.
  Variation: test all pairs of inputs but only for inputs that could influence each other.
- **Random strategy** generates test cases using one of the other strategies and then picks a subset randomly (presumably because original set of test cases is too big).

Heuristic:
1. Each Valid Input at Least Once in a Positive Test Case
2. Test Invalid Inputs Individually Before Combining Them
[To verify the SUT is handling a certain invalid input correctly, it is better to test that invalid input without combining it with other invalid inputs.]

- This is not to say never have more than one invalid input in a test case.
- If you can afford more test cases, also testing with combinations of invalid inputs.